Bien développer pour le Web 2.0

Bonnes pratiques Ajax

Prototype • Script.aculo.us • Accessibilité • JavaScript • DOM • XHTML/CSS

Christophe Porteneuve

Préface de Tristan Nitot,
président de Mozilla Europe



EYROLLES

Bien développer pour le Web 2.0

CHEZ LE MÊME ÉDITEUR

J Battelle, trad. D. Rueff, avec la contribution de S. Blondeel – **La révolution Google**.
N°11903, 2006, 280 pages.

Ouvrages sur le développement web

M. Plasse. – **Développez en Ajax**.
N°11965, 2006, 314 pages.

J. Protzenko, B. Picaud. – **XUL** (coll. Cahiers du programmeur).
N°11675, 2005, 320 pages.

H. Wittenbrik. – **RSS et Atom**. Fils et syndications.
N°11934, 2006, 216 pages.

R. Goetter. – **CSS 2 : pratique du design web** (collection Blanche).
N°11570, 2005, 324 pages.

D. Thomas *et al*. – **Ruby on Rails**.
N°11746, 2006, 590 pages.

T. Ziadé. – **Programmation Python**.
N°11677, 2006, 530 pages.

E. Daspet et C. Pierre de Geyer. – **PHP 5 avancé**.
N°12004, 3e édition 2006, 804 pages.

M. Mason. – **Subversion.** *Pratique du développement collaboratif avec SVN.*
N°11919, 2006, 206 pages.

G. Ponçon. – **Best practices PHP 5.** Les meilleures pratiques de développement en PHP.
N°11676, 2005, 480 pages.

S. Mariel. – **PHP 5 (et XML)** (Les Cahiers du programmeur).
N°11234, 2004, 290 pages.

J. Zeldman. – **Design web : utiliser les standards**, CSS et XHTML.
N°12026, 2e édition 2006, 444 pages.

Autres ouvrages : Web et logiciel libre

S. Blondeel. – **Wikipédia.** *Comprendre et participer.*
N°11941, 2006, 168 pages (collection Connectez-moi !).

F. Le Fessant. – **Le peer-to-peer.** *Comprendre et utiliser.*
N°11731, 2006, 168 pages (collection Connectez-moi !).

C. Béchet. – **Créer son blog en 5 minutes**.
N°11730, 2006, 132 pages (collection Connectez-moi !).

F. Dumesnil. – **Les podcasts.** *Écouter, s'abonner et créer.*
N°11724, 2006, 168 pages (collection Connectez-moi !).

O. Saraja. – **La 3D libre avec Blender**.
N°11959, 2006, 370 pages.

L. Dricot, contrib. de R. Mas. – **Ubuntu**. *La distribution Linux facile à utiliser* (coll. Accès libre).

N°12003, 2e édition 2006, 360 pages avec CD-Rom.

M. Krafft, adapté par R. Hertzog, R. Mas, dir. N. Makarévitch. – **Debian**. *Administration et configuration avancées.*
N°11904, 2006, 674 pages.

S. Gautier, C. Hardy, F. Labbe, M. Pinquier. – **OpenOffice.org 2 efficace**.
N°11638, 2006, 420 pages avec CD-Rom.

C. Gémy. – **Gimp 2 efficace**.
N°11666, 2005, environ 350 pages (collection Accès libre).

M. Grey. – **Mémento Firefox et Thunderbird**.
N°11780, 2006, 14 pages.

Bien développer pour le Web 2.0

Ajax • Prototype • Scriptaculous
Xhtml/Css • JavaScript • Dom

Christophe Porteneuve

Préface de Tristan Nitot,
président de Mozilla Europe

EYROLLES



ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com



Dépôt légal : novembre 2006
N° d'éditeur : 7540
Imprimé en France

À ma mère, Claude Paris.
Tout ça grâce aux premiers pas, tu sais, les petits…

Préface

Vous tenez donc entre vos mains un exemplaire du livre *Bien développer pour le Web 2.0*. On pourrait croire que ce qui est important dans le titre, c'est « Web 2.0 ». Certes, c'est bien le cas. La participation grandissante des utilisateurs, qui est l'une des deux particularités du Web 2.0, est importante. Vitale, même. Paradoxalement, cette notion d'un Web où chacun pourrait à la fois lire et écrire, consommer et produire, est celle de son inventeur, Tim Berners-Lee, même si peu d'internautes ont réalisé cela.

Mais ce qui est surtout important dans le titre de cet ouvrage, c'est « Bien développer ». Développer « comme il faut ». Car le Web dit « 1.0 » ne s'est pas seulement traduit par un Web où seuls quelques auteurs publiaient pour une foule grandissante de lecteurs : le Web 1.0 s'est aussi traduit par des errements technologiques qui ont fait que la promesse du Web n'a été tenue que partiellement, dans la mesure où les exclus étaient nombreux. Vous n'utilisez pas tel plug-in ? Ah, dommage ! Vous avez recours à tel navigateur trop moderne ? Tant pis pour vous ! Vous souhaitez consulter le site avec votre téléphone mobile ? Vous devrez attendre de trouver un PC connecté. Vous avez désactivé JavaScript dans votre navigateur pour des raisons de sécurité ? Passez votre chemin ! Vous avez un handicap visuel ou des difficultés pour manipuler une souris ? Navré, le service n'est pas conçu pour vous. Combien de millions de personnes se sont retrouvées confrontées à de tels problèmes du Web 1.0 ? C'est impossible de le dire... Mais ça n'était pas tant le Web qui était en cause que la mauvaise façon dont les sites ont été développés, souvent par faute de formation, de recul sur la technologie, encore toute récente.

Aussi, alors que le Web 2.0 fait tant parler de lui, qu'il convient d'acquérir les compétences techniques pour construire un site utilisant ces technologies, autant apprendre dès le début la bonne façon de faire. La bonne façon, c'est celle qui consiste à utiliser des méthodes permettant de conserver la compatibilité avec un éventail aussi large que possible de navigateurs, d'utilisateurs, de paramétrages, et de connexions.

Le Web 2.0 fait deux promesses explicites : plus de participation des utilisateurs, et des interfaces plus agréables et ergonomiques. Il en est une autre qui est implicite : que les développeurs web apprennent des échecs et difficultés du Web 1.0 pour ne pas les répéter. Pour éviter le bricolage que fut le Web à ses débuts, en passant à l'époque de la maturité et de l'industrialisation, en permettant un accès à tous. C'est en cela que ce livre est important : il ne s'agit pas seulement d'apprendre à « développer pour le Web 2.0 » mais aussi d'apprendre à bien développer pour le Web.

Tristan Nitot
Président de Mozilla Europe

Table des matières

PREMIÈRE PARTIE

Donner vie aux pages 17

CHAPITRE 2

Ne prenez pas JavaScript pour ce qu'il n'est pas..................... 19

CHAPITRE 7

Une ergonomie de rêve avec script.aculo.us 269

ANNEXE A
Bien baliser votre contenu : XHTML sémantique ... 461

ANNEXE B
Aspect irréprochable et flexible : CSS 2.1 ... 491

Avant-propos

Avant d'entrer dans le vif du sujet, faisons le point sur les domaines abordés dans ce livre, ce qu'il contient et ce que j'ai choisi d'omettre ainsi que sur sa structure générale. Nous verrons notamment comment s'articulent les différents thèmes et l'apprentissage.

À qui s'adresse ce livre ?

Toute personne intéressée de près ou de loin par les technologies web trouvera son intérêt dans cet ouvrage. Précisons néanmoins qu'une connaissance préalable des technologies de contenu web statique est préférable : en l'occurrence, HTML (ou mieux, XHTML) et CSS.

Dans l'idéal, ces connaissances sont « actualisées », et donc conformes aux standards (XHTML Strict, CSS 2.1) et bien maîtrisées, notamment en termes de balisage sémantique. Les lecteurs ayant des lacunes sur ces technologies pourront toutefois trouver, dans les annexes A et B, une présentation succincte des principes fondamentaux, ainsi que de nombreuses ressources – papier ou en ligne – pour parfaire leurs connaissances.

Il n'est par ailleurs pas nécessaire d'avoir des compétences préalables en JavaScript ou DOM, ces sujets étant présentés en détail dans cet ouvrage. En somme, ce livre trouvera son public tant auprès des professionnels chevronnés désireux de se mettre à jour, que des étudiants souhaitant aller au-delà de leurs cours de technologies web, souvent sommaires et trop empiriques, voire obsolètes.

Qu'allez-vous trouver dans ce livre ?

Le livre est découpé en trois parties, précédées de cet avant-propos et d'un chapitre introductif qui présente le Web 2.0 et ses technologies.

Par ailleurs, l'ouvrage est développé sur deux axes forts : un axe thématique et un axe méthodologique et qualitatif. Le premier axe guide le plan, tandis que le second est transversal :

- La première partie présente en détail les technologies qui font « vivre » la page web elle-même, qui sont souvent trop peu ou trop mal connues : JavaScript, DOM et, pour gagner en agilité et en puissance, l'excellente bibliothèque Prototype.
- La deuxième partie explore ce qui fait réellement Ajax, à savoir l'objet `XMLHttpRequest`, moteur de requêtes asynchrones, et les *frameworks* déjà établis dans l'univers du Web 2.0, notamment Prototype et script.aculo.us.
- La troisième partie pousse plus loin la réflexion et l'utilisation en ouvrant vos pages sur des contenus et services externes, au travers des services web, des API REST et des flux de syndication aux formats RSS et Atom.

L'ouvrage est complété par quatre annexes :

- Les annexes A et B fournissent les bases des deux principales technologies de contenu : XHTML et CSS, dans leurs versions récentes.
- L'annexe C constitue **un plus indéniable** : en vous expliquant clairement comment exploiter au mieux les documents de référence qui font le Web (RFC, DTD, recommandations W3C...), elle vous donne accès à une connaissance actualisée et faisant autorité.
- L'annexe D, enfin, quoique placée en fin d'ouvrage, est plutôt à lire d'entrée de jeu : elle donne les clés d'un développement plus productif dans votre navigateur, et vous évite de peiner avec les questions de cache en manipulant les exemples de ce livre.

Dans tous ces chapitres, j'ai été guidé par un souci constant de qualité, tant pour la technique elle-même que pour la méthodologie de travail. Qu'il s'agisse de *unobstrusive JavaScript* (concept que nous étudierons en détail au chapitre 2), de balisage sémantique, de CSS efficaces, d'accessibilité, ou du bon choix de format pour un flux de syndication ou le résultat d'une requête Ajax, j'ai fait de mon mieux pour vous permettre de réaliser un travail haut de gamme, différencié, et pour tout dire, constituant un fameux avantage compétitif, à l'heure où tout un chacun n'hésite pas à clamer sur son CV qu'il est un « développeur web expert ».

ADRESSAGE « je » ou « nous » ?

Je me suis posé la question du pronom employé lorsque l'auteur est sujet des phrases. D'aucuns dirons que « nous » est plus pudique, plus modeste, plus convenable. C'est possible, mais ce livre, je l'écris pour vous, et il m'est très agréable de vous imaginer en train de le lire. D'ailleurs, j'aime à croire qu'il vous plaira et, surtout, vous sera utile. Et puis s'il ne vous plaît pas, c'est *ma* faute, pas *la nôtre*. Alors, par simplicité, par convivialité, « je ».

Les standards du Web

Tout le monde parle en bien des standards du Web, et affirme qu'il vaut mieux les respecter. Mais personne ne dit vraiment de quoi il s'agit, pourquoi c'est préférable, et vers où se tourner pour mettre le pied à l'étrier.

À l'heure où une portion significative des développeurs web français chevronnés, et la majorité des jeunes diplômés français croyant « connaître le développement web », ignorent ce qu'est le W3C, ne savent pas donner la dernière version de HTML, sont dans le flou sur les différences exactes entre XHTML et HTML, et pensent que CSS se résume à coller des balises `div` et des attributs `class` et `style` partout, je mesure le chemin à parcourir en termes d'évangélisation et d'éducation en général.

De quelles technologies parle-t-on ?

Commençons par passer en revue les technologies qui font aujourd'hui figure de standards du Web. Je restreindrai la liste aux technologies qui se rapprochent de mes propos, sous peine d'y consacrer de nombreuses pages.

- **HTML** (*HyperText Markup Language*) est le langage établi de description de contenus dans une page web. Dérivé du SGML, sa syntaxe est un peu trop permissive pour éviter toute ambiguïté et permettre un traitement automatisé vraiment efficace.
- **XML** (*eXtensible Markup Language*) est une syntaxe plus formelle de balisage de contenus, qui garantit le traitement automatique du document sans risquer ni ambiguïtés, ni soucis de jeux de caractères, ni limitations de types ou tailles de contenu.
- **XHTML** revient essentiellement à appliquer à HTML les contraintes syntaxiques de XML, ouvrant ainsi la porte au traitement fiable du contenu des pages web.
- **CSS** (*Cascading Style Sheets*, généralement « feuilles de styles » en français) est une technologie de présentation permettant une mise en forme extrêmement

avancée des contenus compatibles XML (mais aussi au HTML, par souci de flexibilité). Les possibilités sont énormes, bien au-delà de ce que permettaient les quelques balises « présentation » de HTML.

- **DOM** (*Document Object Model*) décrit une série d'outils à destination des programmeurs (on parle d'*interfaces*) permettant de représenter et de manipuler en mémoire un document compatible XML. Ces manipulations sont pratiquement sans limites, et constituent un des piliers d'une page web « vivante ». Parmi les sous-parties de DOM, on citera notamment *Core*, qui fournit le noyau commun à tous les types de documents ; *HTML*, spécialisé dans les pages web ; et enfin *Events*, qui gouverne le traitement des événements associés aux éléments du document.
- **JavaScript** est un langage de script, dynamique, orienté objet et disposant de nombreuses fonctions avancées, aujourd'hui disponible sous une forme ou sous une autre dans tous les navigateurs un tant soit peu répandus. Sans lui, pas de pages vivantes, pas de Web 2.0, pas d'Ajax !
- **XMLHttpRequest** est un objet capable d'envoyer des requêtes asynchrones via HTTP (voilà une phrase qui ne vous dit peut-être pas grand-chose ; pas d'inquiétude, le chapitre 5 vous éclairera bientôt). Utilisé en JavaScript, il constitue le cœur d'Ajax.
- **RSS** 1.0 (*RDF Site Summary*) est un format de flux de syndication, défini de façon beaucoup plus formelle que ses homonymes de versions 0.9x ou 2.0 (où l'abréviation signifie *Really Simple Syndication*), lesquels sont plus répandus mais moins puissants. Il est basé sur RDF (*Resource Description Framework*), grammaire formelle de représentation de la connaissance autour de laquelle gravite l'univers du Web sémantique (pour plus de détails sur le sujet, consultez par exemple http://www.w3.org/2001/sw/).
- **Atom** est le format de flux de syndication le plus récent, sans doute le plus puissant et le plus efficace aussi, sans pour autant verser dans la complexité.

Parmi les standards du Web, on trouve encore de nombreuses technologies très employées, comme PNG (format d'image), SOAP et les services Web, XSL et XSLT ; ainsi que d'autres encore trop rarement employées, par exemple SVG (images vectorielles), MathML (formules mathématiques), SMIL (multimédia), XForms (pour des formulaires web très performants)...

Qui est à la barre, et où va-t-on ?

Ces standards ne s'inventent pas tout seul ; à leur origine, on trouve le plus souvent une organisation, un comité ou une association, parfois une entreprise, plus rarement encore un individu. Mais ceux qui ont fait naître une technologie n'en assurent pas toujours bien l'évolution, comme c'est le cas pour HTML.

Comprendre qui s'occupe d'un standard permet de savoir où suivre son évolution, déterminer à quoi s'attendre dans les années à venir, et mieux comprendre ses orientations et les choix qui le gouvernent.

- **(X)HTML** a été principalement maintenu par le W3C (*World Wide Web Consortium*), groupement international d'associations, d'entreprises et d'individus en charge de la plupart des technologies du Web, essentiellement dans le contexte des navigateurs. Hélas, après avoir publié HTML 4.01 en 1999, le W3C a délaissé HTML pour se concentrer sur le Web sémantique, CSS et les technologies gravitant autour de XML.
  Le problème, c'est que HTML est l'outil fondamental de tout développeur web, quelle que soit la technologie côté serveur utilisée, Ajax ou non, Web 2.0 ou non. Sans évolution depuis le début du siècle, il échoue à satisfaire nombre de besoins récurrents.
  Devant la difficulté à remobiliser le W3C autour de HTML, un groupement séparé a vu le jour, le WHAT WG (*Web Hypertext Application Technology Working Group*, http://whatwg.org). Constitué principalement de figures de proue des standards, presque tous par ailleurs membres du W3C, il jouit déjà d'une excellente notoriété et d'une large approbation. Il vise à mettre au point plusieurs standards, dont deux souvent désignés par dérision sous l'appellation commune « HTML 5 » : Web Applications 1.0 et Web Forms 2.0. Ces deux projets augmentent énormément les possibilités pour le développeur web, et plusieurs navigateurs de premier plan ont annoncé leur intention de les prendre en charge...
- **CSS** est également l'œuvre du W3C, dont il reste un cheval de bataille important. Depuis la version 2, remontant à 1998 (!), le standard évolue de façon double.
  D'un côté, une version 2.1 est en chantier permanent (à l'heure où j'écris ceci, la dernière révision date d'avril 2006) et constitue une sorte de correction de la version 2, qui en précise les points ambigus, ajoute quelques compléments d'informations, etc.
  De l'autre, la version 3 est un chantier proprement pharaonique, à tel point que le standard est découpé en pas moins de 37 modules. Parmi ceux-là, certains font l'objet de beaucoup d'attentions, et sont au stade de la *recommandation candidate* (dernière étape avant l'adoubement au rang de standard), ou de dernier appel à commentaires. Il ne s'agit au total que de 11 modules sur 37. Pour les autres, soit le travail n'a carrément pas démarré, soit ils disposent d'une ébauche qui, parfois, stagne pendant des années (le module de gestion des colonnes, pourtant réclamé à corps et à cris par beaucoup, a ainsi gelé entre janvier 2001 et décembre 2005 !).
  Enfin, même si CSS a d'ores et déjà révolutionné la création de pages web, on verra qu'il existe un gouffre entre les dernières versions et l'état de l'art dans les navigateurs...

- **DOM** est aussi à la charge du W3C. En DOM, on ne parle pas de versions mais de niveaux. Le W3C travaille régulièrement dessus au travers de ses sous-projets : *Core*, *HTML*, *Events*, *Style*, *Views*, et *Traversal and Range*.
- **JavaScript** a été inventé en 1995 par Brendan Eich pour le navigateur Netscape Navigator 2.0. C'est aujourd'hui l'ECMA, organisme international de spécifications, qui gère son évolution au travers des diverses éditions du standard ECMA-262. Brendan Eich continue à piloter la technologie et travaille aujourd'hui pour Mozilla.
- **XMLHttpRequest** a été inventé par Microsoft pour Internet Explorer (MSIE) 5.0. Depuis 2002, des équivalents ont fait leur apparition dans la plupart des navigateurs, au point qu'un standard W3C est en cours de rédaction pour enfin ouvrir totalement la technologie.
- **RSS** est un sigle qui masque en réalité deux technologies bien distinctes. La première version historique, la 0.90, vient de Netscape (1999). Les versions 0.9x suivantes et 2.0 sont l'œuvre du seul Dave Winer, et fournissent une solution simple (et même simpliste) aux besoins les plus courants de la syndication de contenu. Il s'agit de standards gelés, qui n'évolueront plus. La version 1.0 est beaucoup plus puissante, mais aussi plus complexe, car basée sur RDF, donc sur un standard formel lourd réalisé par le W3C. Elle est encore, pour l'instant, moins utilisée que ses homonymes.
- **Atom** a été défini, contrairement à RSS, dans la stricte tradition des standards Internet : au moyen d'un forum de discussion ouvert, et encadré dès le début par l'IETF, organisme international chargé de la plupart des protocoles Internet (comme HTTP). Il gagne sans cesse en popularité, et constitue très officiellement un standard (ce qu'on appelle une RFC) depuis décembre 2005, sous le numéro 4287 (http://tools.ietf.org/html/rfc4287).

Les principaux acteurs des standards du Web sont donc le W3C et, sans doute de façon moins visible pour l'utilisateur final, l'IETF. On constate néanmoins que le premier est parfois prisonnier de sa propre bureaucratie, au point que des groupes externes reprennent parfois le flambeau, comme c'est le cas autour de HTML avec le WHAT WG.

Ce panorama ne serait pas complet sans évoquer le WaSP (*Web Standards Project*, http://webstandards.org), véritable coalition d'individus ayant appréhendé tout l'intérêt des standards du Web et la portée de leur application. Ce groupe fut un acteur important de l'arrêt de la « guerre des navigateurs » qui a fait rage dans les années 1990, laissant Navigator sur le carreau et faisant entrer MSIE dans la léthargie qu'on lui connaît.

Mais surtout, il œuvre sans relâche pour rallier toujours plus d'acteurs, notamment les éditeurs commerciaux, à la prise en charge des standards. En collaborant avec

Microsoft, mais aussi Adobe et Macromedia (du temps où ils n'avaient pas fusionné), ainsi que de nombreux autres, le WaSP aide à rendre les produits phares du marché plus compatibles avec les standards et l'accessibilité. Vraiment, grâce leur soit rendue ! Sans eux, on en serait encore à devoir annoter la moindre mention technique dans un ouvrage comme celui-ci à coups de « IE seulement », « NN seulement », « non supporté », etc.

Quels sont donc ces avantages extraordinaires qui ont convaincu tant de volontaires de fonder ou rejoindre le WaSP, et de partir en croisade auprès des éditeurs ? C'est ce que je vais vous expliquer dans la section suivante.

À quoi servent les standards ?

Encore aujourd'hui, on rencontre de nombreuses personnes qui, lorsqu'on évoque les standards du Web, rétorquent quelque chose comme : « et alors ? Je n'utilise pas tout ça et mon site marche ! Pourquoi devrais-je faire autrement ? ».

Il s'agit là d'une vue très étroite du site en question. Sans vouloir vexer personne, cela revient à ne pas voir plus loin que le bout de son nez, à ne se préoccuper que de soi et de son environnement immédiat, ce qui est tout de même inattendu pour un contenu censé être accessible par le monde entier, souvent pour longtemps.

L'expression désormais consacrée HTML des années 1990 est utilisée pour désigner ce mélange d'habitudes techniques aujourd'hui dépassées : balisage hétéroclite mélangeant allègrement forme et fond, utilisant à mauvais escient certaines balises (ce qu'on appelle de la « soupe de balises ») ; emploi inapproprié ou incohérent de CSS ; surabondance d'éléments `div` ou d'attributs `class` superflus (syndromes baptisés *divitis* et *classitis*) ; déclinaisons manuelles ou à peine automatisées des pages suivant les navigateurs visés ; et bien d'autres usages que je ne saurai tous citer ici.

Cette façon de faire, fruit d'une approche fondamentalement empirique du développement web et d'une évolution souvent organique des sites, sans plan cohérent préalable, était peut-être inévitable pour la première génération du Web. Après tout, la première version d'un projet contient souvent de nombreuses horreurs qu'il faudra éliminer. Mais il ne s'agit pas ici que d'esthétique. Les conséquences pénibles de cette approche sont nombreuses, et accablent aujourd'hui encore un grand nombre de projets et sociétés qui persistent à ne pas évoluer :

- Faute d'une utilisation intelligente de CSS et de JavaScript, les pages sont beaucoup trop lourdes, constituées pour 10 % ou moins de contenu véritable. Impact : le coût élevée de la bande passante pour votre site. Pour un site très visité (à partir du million de visiteurs uniques par mois), le superflu atteint un tel volume que son coût se chiffre fréquemment en dizaines voire centaines de milliers d'euros par mois.

- Un balisage lourd ou rigide, ainsi qu'un emploi inadapté de CSS, amènent aussi à des pages ne pouvant pas fonctionner telles quelles sur différents navigateurs, sans parler des modes de consultation alternatifs, toujours plus répandus : assistant personnel (PDA), téléphone mobile 3G, borne Internet sans souris dans un espace public, *Tablet PC*, impression papier pour lecture ultérieure, et j'en passe. Pour toucher une plus vaste audience, il faut donc mettre en place des versions spécialisées de chaque page, travail particulièrement fastidieux aux conséquences néfastes : d'une part cela multiplie l'espace disque nécessaire, d'autre part ce travail est généralement réalisé sommairement, de sorte que certaines versions des pages sont de piètre qualité, ou ne sont pas mises à jour assez souvent.
- Une mauvaise utilisation des CSS entraîne généralement une intrusion de l'aspect dans le contenu, et rend l'apparence des pages difficile à changer globalement. Toute refonte de la charte graphique d'un site devient vite un cauchemar, à force de devoir dénicher tous les styles en ligne et les balises de mise en forme restées cachées au fond d'une page.
- À moins d'avoir été profondément sensibilisé à la question, une équipe de conception et développement de sites web aura tendance à enfreindre à tour de bras les règles d'or de l'accessibilité. Non seulement les pages seront difficilement exploitables par les non-voyants, les malvoyants, les personnes souffrant d'un handicap, même léger, rendant impossible l'utilisation de la souris, mais aussi par les programmes de traitement automatique, comme Google. En effet, les sites peu accessibles sont souvent beaucoup moins bien classés dans les moteurs de recherche que ceux qui respectent les principes fondamentaux d'accessibilité. N'oublions pas que le Web est censé, par définition, être accessible par tous. Dans *World Wide Web*, il y a *World*.

À l'inverse, modifier ses méthodologies de travail pour accéder à un niveau supérieur de qualité, à une façon plus actuelle, et finalement plus facile de réaliser des sites web, produit rapidement des bénéfices :

- Un site séparant clairement le contenu (balisage XHTML) de la forme (feuilles CSS) et du comportement (scripts JS, c'est-à-dire JavaScript) produira nécessairement des pages infiniment plus légères, sans parler de l'efficacité accrue des stratégies de cache des navigateurs devant un découpage des données en fichiers distincts. Après avoir refondu complètement son site, ESPN, principale chaîne de sports aux États-Unis, a augmenté son audience tout en divisant à tel point ses coûts de bande passante que l'économie mensuelle, malgré un tarif extrêmement avantageux, se chiffrait en dizaines de milliers de dollars ! (Pour davantage de détails voir l'adresse suivante : http://www.mikeindustries.com/blog/archive/2003/06/espn-interview).
- Une utilisation appropriée des CSS implique d'avoir une seule page XHTML. J'insiste : *une seule*. Si vous croyez encore qu'il s'agit d'un mythe, allez donc faire un

tour sur le CSS Zen Garden (http://www.csszengarden.com). Lorsque vous aurez essayé une petite vingtaine de thèmes, réalisez que la seule chose qui change, c'est la feuille de styles. La page HTML est strictement la même. Il ne s'agit pas ici seulement d'offrir des thèmes, mais bien d'offrir une vue adaptée de la page pour de nombreux usages et périphériques de consultation : page agréable à l'impression, mais aussi sur un petit écran (on pense aux PDA et aux téléphones mobiles), ou avec des modes spéciaux pour les malvoyants (comme un contraste fort, un fond noir, etc.). Puisqu'il n'y a qu'une seule page, elle est fatalement à jour, et les versions alternatives sont donc sur un pied d'égalité. Tout en gagnant de l'audience, vous la traitez mieux en lui garantissant le même contenu pour tous.

- Une prise en compte systématique de l'accessibilité, qui elle non plus n'entrave en rien la page, facilite la vie aux utilisateurs accablés d'un handicap quelconque (plus de 20 % des internautes aux États-Unis et en France, selon certaines études). Couplée à l'emploi d'un balisage sémantique, elle signifie aussi que l'indexation de la page par les moteurs de recherche sera de bien meilleure qualité, augmentant votre visibilité et donc vos revenus potentiels.

Mercantilisme… aveugle ?

Le patron qui éructait, lors d'une conférence, « on vend des écrans plasma, on s'en fout des aveugles, ce ne sont pas nos clients ! » s'est vu gratifier d'une réponse à l'évidence cinglante : le malvoyant voire non-voyant n'en est pas moins internaute, et s'il ne peut offrir à un proche un bel écran acheté sur votre site, il l'achètera ailleurs. La réflexion vaut, évidemment, pour tous les handicaps…

Qu'en pensent les concepteurs de navigateurs ?

Disons qu'ils sont partagés. Et pour être précis, c'est un peu une situation « un contre tous ». D'un côté, on trouve les navigateurs libres accompagnés de quelques navigateurs commerciaux traditionnellement respectueux des standards : Mozilla, Firefox et Camino, Konqueror, Opera et Safari, pour ne citer qu'eux. De l'autre, on trouve un navigateur commercial qui, dès qu'il n'a plus eu de concurrence significative à partir de 1999, a sombré dans la léthargie, j'ai nommé MSIE.

La situation n'est toutefois pas si claire : tous les navigateurs n'ont pas le même niveau de prise en charge pour tous les standards, et le travail sur MSIE a repris en vue d'une version 7. Dressons ici un rapide portrait des principaux navigateurs au regard des standards. Gardez à l'esprit que cette situation évolue rapidement, et que vous aurez intérêt à suivre l'actualité des principaux navigateurs pour vous tenir à jour.

Notez que tous les navigateurs ci-dessous supportent `XMLHttpRequest`. MSIE utilise encore un ActiveX qui deviendra un objet natif JavaScript standard dans IE7, tandis que les autres navigateurs utilisent déjà un objet natif JavaScript.

Mozilla/Firefox/Camino	
Ils utilisent peu ou prou le même moteur, même si la suite Mozilla a été abandonnée par la fondation, et que sa mise à jour est désormais assurée par une communauté de volontaires, qui peuvent parfois mettre du temps à intégrer les nouveautés de Firefox dans la suite complète (projet Seamonkey). Quant à Camino, c'est un Firefox spécialisé Mac OS X, globalement équivalent côté standards. On parle ici de Firefox 1.5.	
(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Bon support de CSS 2.1, hormis quelques aspects encore exotiques, et support émergeant de certains modules 3.0.
JavaScript	Naturellement le meilleur, puisque le chef d'orchestre de la technologie travaille pour la fondation Mozilla. Toujours à jour sur la dernière version. Firefox 1.5 supporte JS 1.6 et la version 2.0 supportera JS 1.7.
DOM	Très bon support du niveau 2, support partiel du 3.

Safari 2	
(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Très bon support de CSS 2.1, hormis quelques aspects encore exotiques (notamment les styles audio).
JavaScript	Bon support de JS 1.5.
DOM	Bon support du niveau 2.

Opera 9	
À noter qu'Opera propose une excellente page pour suivre sa compatibilité aux standards : http://www.opera.com/docs/specs/	
(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Excellent support de CSS 2.1 (passe le test Acid2).
JavaScript	Prend en charge ECMA-262 3rd, soit JS 1.5.
DOM	Très bon support du niveau 2, bon support du 3.

Konqueror 3.5.2	
(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Excellent support de CSS 2.1, (passe le test Acid2), support partiel de CSS 3.
JavaScript	Bon support de JS 1.5.
DOM	Très bon support du niveau 2, support partiel du 3.

Internet Explorer 6	
(X)HTML	Très bon support, à hauteur des versions récentes. Un souci avec les prologues XML, sans grande importance.
CSS	Support de CSS 1, très partiel de CSS 2.
JavaScript	Jscript 6.0, pas totalement compatible JavaScript, fonctionnellement entre JS 1.3 et 1.5.
DOM	Support correct du niveau 2, mais persiste à se comporter parfois différemment du standard !

Internet Explorer 7 (projections sur annonces)	
(X)HTML	Plus de souci de prologues.
CSS	Support a priori total de CSS 1 et correct de CSS 2.1.
JavaScript et DOM	Pratiquement aucune amélioration prévue, ni en version, ni en vitesse. Entre le DOM des objets `select`, le `getElementById` qui utilise aussi l'attribut `name`, les prototypes non modifiables des objets natifs, le modèle événementiel propriétaire (notamment pas de `addEventListener`) ou l'absence très pénible de DOM niveau 3 XPath, il y a de quoi faire pour IE8....

Comme on peut le constater, MSIE reste loin derrière les autres. Il ne faut pas s'étonner si Konqueror gagne du terrain chez les utilisateurs de Linux, et si Firefox a grignoté, en à peine 2 ans, plus de 20 % de parts de marché, et jusqu'à 37 % dans certains pays. Rien qu'en France, pourtant au 17e rang en Europe, les 18 % représentent quelque 4,8 millions d'internautes. Quiconque continue à développer un site Internet au mépris des standards ferait mieux de ne pas avoir trop d'ambitions commerciales...

Quelques mots sur les dernières versions

Voici un rapide tour d'horizon des versions en cours et à venir pour les principaux standards. Là aussi, un peu de veille sera votre meilleur atout.

- **HTML est en version 4.01** (décembre 1999). Son évolution passera sans doute par le « HTML 5 » du WHAT WG, mais la prise en charge par les navigateurs est balbutiante (Web Forms 2.0 en bêta dans Opera 9 et sous forme d'extension pour Firefox, par exemple).
- **XHTML est en version 1.1**. La plupart des navigateurs implémentent au moins la version 1.0. La version 1.1 est plus stricte et demande normalement un type MIME distinct, qui panique notamment MSIE pour le moment ! En revanche, certains aspects de la prochaine version, la 2.0, sont dénigrés par le plus grand nombre, au motif principal qu'elles cassent vraiment trop la compatibilité descendante sans grand avantage en retour.
- **CSS est en version 2.1**, avec beaucoup de travail autour des 37 modules composant CSS 3.0. Le calendrier de sortie de ces modules à titre de recommandations

s'étalera probablement sur au moins 5 ans… Là où Konqueror, Safari et Opera sont plutôt au en tête, Firefox a pour l'instant un tout petit peu de retard, et MSIE est très, très loin derrière. Toutefois, son imminente version 7 a fait d'immenses progrès là-dessus.

- **DOM est au niveau 2**, et avance bien au niveau 3, plusieurs modules étant terminés, dont le *Core*. La plupart des navigateurs s'attaquent fortement à ce nouveau niveau, et même MSIE devrait rattraper un peu son retard prochainement.
- **JavaScript est en version 1.7**, actuellement uniquement pris en charge par Firefox 2.0, mais la disponibilité de bibliothèques Java et C++ toutes prêtes (comme Rhino) facilitent l'intégration par d'autres navigateurs. La version 1.7 apporte quelques grandes nouveautés, mais ce n'est rien à côté de la version 2.0, qui devrait sortir au 2e trimestre 2007, après une version 1.9 au premier trimestre.

Qu'est-ce que le « Web 2.0 » ?

Le terme « Web 2.0 », qui a envahi la presse et les sites spécialisés, décrit en réalité deux phénomènes distincts.

D'une part, il y a cette évolution profonde des interfaces utilisateur proposées en ligne, qui rattrapent en convivialité et en interactivité celles qu'on trouve sur des applications plus classiques (applications dites « desktop », au sens où elles s'exécutent en local sur la machine de l'utilisateur), ou même sur celles qui équipent des périphériques légers (téléphones mobiles, assistants personnels, etc.).

Glisser-déplacer, complétion automatique, création dynamique d'images, personnalisation à la volée de l'interface, exécutions en parallèle : autant de comportements que nous avons pris l'habitude de trouver dans les applications, et qui manquaient cruellement – jusqu'à récemment – aux navigateurs. Ceux-ci étaient réduits à des rôles subalternes, à un sous-ensemble ridiculement étriqué de possibilités bien établies. Et pourtant, les navigateurs ne sont pas plus bêtes que les autres programmes : nous les avons simplement sous-exploités jusqu'ici.

L'autre facette du Web 2.0, cest ce qu'on pourrait appeler « le Web aux mains des internautes ».

Il n'y a pas si longtemps, consulter une page web constituait une expérience similaire à lire une page imprimée dans un magazine : on n'avait pas son mot à dire sur l'aspect. Cette barre de navigation sur la droite vous gâche-t-elle la vue ? Ce bandeau de publicité vous énerve-t-il ? Le texte est-il trop petit, ou le contraste trop faible pour votre vue ? Tant pis pour vous ! Le concepteur graphique du site l'a voulu ainsi, et sa volonté fait loi. En fait, consulter une page web était encore pire que lire un

magazine : sur ce dernier, au moins, on bénéficiait de l'excellente résolution de l'impression, de sorte que les textes en petite taille étaient bien plus lisibles. Bien sûr, la plupart des navigateurs permettent de désactiver CSS ou d'utiliser une feuille de styles personnelle, ou encore de zoomer sur le texte voire sur toute la page (images comprises), mais c'est une piètre consolation.

Et voilà que de nouveaux usages apparaissent, qui donnent enfin à l'internaute la haute main sur l'aspect final de la page sur son navigateur. Exit, les parties superflues et irritantes ! Agrandi, le texte principal écrit bien trop petit ! Et tant qu'à faire, augmentons la marge entre les paragraphes et aérons le texte en changeant l'interligne ! À l'aide d'outils dédiés, tels que les extensions GreaseMonkey (http://greasemonkey.mozdev.org) et Platypus (http://platypus.mozdev.org) pour Firefox, le visiteur peut ajuster comme bon lui semble l'aspect d'une page, et rendre ces ajustements automatiques en prévision de ses visites ultérieures.

Par ailleurs, les internautes peuvent maintenant contribuer à faire l'actualité du Web, tant grâce à la facilité de publication qu'offrent des outils comme les blogs, qu'au travers d'annuaires de pages très dynamiques basés sur des votes de popularité (par exemple, Digg). Le principe est simple : ces annuaires permettent à tout un chacun de « voter » pour une page quelconque du Web. Les annuaires maintiennent alors une liste, par popularité décroissante, des pages ainsi signalées.

Si un nombre massif d'internautes votent pour une même page, celle-ci apparaît fatalement en excellente position dans l'annuaire qui a recueilli les votes. Le résultat net est séduisant : les annuaires en haut de liste ont un contenu qui a intéressé, amusé ou marqué un maximum de gens. Statistiquement, il a donc toutes les chances de vous intéresser, vous aussi.

Sous un angle plus politique, cela signifie que les « gros titres » ne sont plus confiés à une salle de rédaction, si facile à instrumentaliser. Pour acquérir une telle visibilité, fut-elle éphémère, la page n'a d'autre choix que de plaire à beaucoup de monde. C'est un système très démocratique.

Les principaux sites de ce type : del.icio.us (http://del.icio.us/) et Digg (http://www.digg.com, plus orienté technologies) pour n'en citer que deux, sont déjà extrêmement visités (plusieurs dizaines de millions de visiteurs uniques par jour). Du coup, de nombreux blogs, magazines en ligne et autres sites au contenu très dynamique affichent systématiquement sur leurs pages des liens graphiques aisément reconnaissables pour faciliter (et donc encourager) le vote de l'internaute auprès des principaux annuaires.

Les sites Technorati et del.icio.us figurent également parmi les pionniers d'un nouvel usage qui se répand rapidement : le *tagging*. Il s'agit de permettre aux internautes de qualifier une page à coup de mots-clés, pour obtenir un système riche de références

croisées et de catégories tous azimuts, bien plus souple que les hiérarchies de catégories habituelles. Certains outils de blog, comme Typo (http://typosphere.org) ou Dotclear 2 (http://www.dotclear.net), proposent déjà l'affectation de *tags* (étiquettes) aux billets.

Le Web 2.0, tout comme Firefox à sa sortie, vous invite finalement à « reprendre la main sur le Web ! »

Vue d'ensemble, chapitre par chapitre

Pour finir cet avant-propos (un peu long, je vous l'accorde), je vous propose de jeter un coup d'œil général à la structure de l'ouvrage, en soulignant son articulation et le rôle de chaque chapitre.

- Le chapitre 1, *Pourquoi et comment relever le défi du Web 2.0 ?*, pose la problématique et les enjeux. Il s'agit de bien saisir la charnière entre les sites classiques et le Web 2.0 ; après de nombreux exemples illustrés et le positionnement des principales technologies dans l'architecture globale d'un développement, le chapitre démystifie Ajax et termine en dressant un plan d'actions, autour de cet ouvrage, pour vous aider à tirer le maximum de bénéfices de votre lecture.

Première partie : donner vie aux pages

Trois chapitres visent à s'assurer que vous maîtrisez bien les piliers désormais classiques sur lesquels se construit aujourd'hui Ajax. À moins que vous ne soyez véritablement un expert en JavaScript et DOM, parfaitement respectueux des standards qui les gouvernent, je ne saurais trop vous recommander de ne pas faire l'impasse sur ces chapitres, au seul prétexte que vous croyez les connaître. En toute probabilité, vous allez y apprendre quelque chose.

- Le chapitre 2, *Ne prenez pas JavaScript pour ce qu'il n'est pas*, présente en détail ce langage si mal connu, accablé d'a priori et souvent bien mal employé. Ce chapitre est très riche en conseils et astuces méthodologiques, et prend soin de vous aider à réaliser une couche « comportement » la plus propre et la plus élégante possible. Une place particulière est accordée au débogage.
- Le chapitre 3, *Manipuler dynamiquement la page avec le DOM*, nous ouvre les voies royales qui mènent aux pages véritablement dynamiques, dont le contenu évolue rapidement, entièrement côté client. De nombreux exemples pour des besoins concrets sont réalisés. Des conseils précieux et un point sur les problèmes résiduels de compatibilité terminent ce chapitre.
- Le chapitre 4, *Prototype : simple, pratique, élégant, portable !*, présente la quasi-totalité de Prototype, sans doute la plus utile des bibliothèques JavaScript largement

répandues. Grâce à elle, nous allons apprendre à réaliser du code JavaScript qui, alors qu'il est parfois plus portable que nos précédentes tentatives, est néanmoins plus élégant, plus concis, plus expressif, et d'une façon générale tellement plus agréable à écrire.

Deuxième partie : Ajax ou l'art de chuchoter

Une fois établies de bien solides et confortables bases techniques, vous allez pouvoir vous plonger dans ce qui constitue, pour beaucoup, la partie la plus visible d'Ajax : les requêtes asynchrones en arrière-plan. C'est grâce à elles que nos pages semblent enfin capables de faire « plusieurs choses en même temps », et n'ont plus autant besoin de se recharger intégralement.

Le chapitre 5, *Les mains dans le cambouis avec XMLHttpRequest*, vous emmène jusqu'aux tréfonds de la technologie responsable des requêtes asynchrones. C'est l'occasion de découvrir une autre technologie de pointe, très agréable elle aussi : le langage Ruby, dont nous nous servirons pour créer, avec une déconcertante facilité, un serveur web à contenus dynamiques pour nos tests.

Le chapitre 6, *Ajax tout en souplesse avec Prototype*, nous fait passer à la vitesse supérieure ! Puisque nous maîtrisons désormais les rouages, nous allons pouvoir délaisser le cambouis pour faire des bonds spectaculaires en productivité avec les facilités Ajax de Prototype (encore lui !).

Le chapitre 7, *Une ergonomie de rêve avec script.aculo.us*, explore l'incroyable bibliothèque d'effets visuels et de comportements avancés proposée par script.aculo.us. Ce chapitre vous emmène par ailleurs plus loin dans la réflexion, autour des usages pertinents ou malvenus d'Ajax et des limites de son utilisation.

Troisième partie : parler au reste du monde

C'est un peu la partie bonus, qui va au-delà de la technologie Ajax pour explorer des usages concrets, et de plus en plus fréquents. L'idée, c'est que nos pages n'ont aucune raison de se limiter à notre serveur et peuvent discuter tout aussi aisément avec n'importe quel site, et n'importe quel service prévu à cet effet.

Le chapitre 8, *WebServices et REST : nous ne sommes plus seuls*, illustre cette idée en présentant ces deux technologies pour s'attacher ensuite à faire profiter nos pages des possibilités de recherche d'Amazon, de prévision de *The Weather Channel*, et des bibliothèques d'images de Flickr.

Le chapitre 9, *L'information à la carte : flux RSS et Atom*, présente les deux principaux formats de flux pour mettre en œuvre une syndication de contenus (blogs et autres) directement sur nos pages.

Des annexes pour le débutant comme pour l'expert

Sur quatre annexes, deux visent à aider le lecteur auquel manqueraient quelques bases, tandis que les deux dernières donnent à tous des compétences recherchées.

L'annexe A, *Bien baliser votre contenu : XHTML sémantique*, rappelle les bases du XHTML et insiste lourdement sur l'importance d'un balisage non seulement valide, mais surtout sémantique. Après avoir succinctement listé les balises pour mémoire, elle fournit également quelques cas concrets de balisage impeccable, correspondant à des besoins récurrents.

L'annexe B, *Aspect irréprochable et flexible : CSS 2.1*, joue le même rôle vis-à-vis de CSS, et donc de la mise en forme. Le vocabulaire est précisé, avant d'attaquer suffisamment les fondamentaux : structure des règles, principe de cascade et modèle de boîtes. Une liste concise des sélecteurs et propriétés permet de ne pas trop patauger dans les exemples du reste de l'ouvrage.

L'annexe C, *Le plus de l'expert : savoir lire une spécification*, apporte un réel plus en vous apprenant à lire les principaux formats de spécification pour les standards du Web, et à naviguer au sein de ces documents parfois complexes, qui font souvent appel à des syntaxes particulières. Être à l'aise avec ces documents présente de nombreux avantages et constitue une compétence encore trop rare.

L'annexe D enfin, *Développer avec son navigateur web*, fait le point sur les possibilités riches ou moins riches pour votre productivité de développeur web sur les principaux navigateurs : gestion du cache, extensions, outils complémentaires de débogage et de test, y sont passés en revue. À lire impérativement, en fait, avant de démarrer le livre !

J'ai fait de mon mieux pour que vous retrouviez dans cet ouvrage autant d'informations techniques, concrètes et de qualité, que ce que je fournis à mes étudiants dans mes cours.

Aller plus loin...

On ne le répétera jamais assez, tout l'ouvrage tente de répondre à un souci constant de qualité, d'élégance, d'efficacité, au travers de nombreux conseils méthodologiques et choix techniques savamment orientés. L'objectif n'est rien moins que vous rendre meilleur que la compétition !

Dans le même esprit, la plupart des chapitres se terminent par une section « Pour aller plus loin... », qui liste des ouvrages et ressources en ligne de qualité permettant d'approfondir les sujets explorés.

À propos des exemples de code

L'ensemble des codes source de ce livre est disponible dans une archive disponible sur le site web des éditions Eyrolles (www.editions-eyrolles.com), accessible depuis la page de l'ouvrage. Certains chapitres n'utilisent que de courts extraits (par exemple, le chapitre 4 sur Prototype), mais l'archive fournit toujours des pages de test complètes.

Ces exemples ont tous été testés sur Firefox 1.5, Safari 2, MSIE 6, Opera 9 et Konqueror 3.5.2. Lorsque certaines contraintes sont incontournables, l'impact est précisé dans le texte du livre.

Par ailleurs, les bibliothèques Prototype et script.aculo.us qui y figurent sont parfois plus récentes que leur dernière version stable publique. Pour Prototype notamment, c'est la version 1.5.0_rc1, avec le patch pour script.aculo.us 1.6.4, que j'ai retenue.

C'est cette version qui est documentée au chapitre 4. Elle a en outre l'avantage de corriger un problème de positionnement sur Opera et un autre sur MSIE, qui auraient causé des soucis pour certains exemples. L'archive des codes source vous fournit aussi ces versions à part, dans un répertoire `bibliotheques_seules`, situé à la racine de l'archive.

Remerciements

Ce livre n'aurait pas vu le jour sans la confiance que m'ont témoignée Muriel Shan Sei Fan et Éric Sulpice. Leur bonne humeur, leur amour de l'informatique et leur dynamisme m'ont d'abord donné envie d'écrire pour Eyrolles, et par la suite grandement facilité la tâche. Un gros merci à Muriel, notamment pour avoir fait sentir très tôt le besoin d'un *extreme makeover* sur la table des matières !

Xavier Borderie et Richard Piacentini ont eu la gentillesse d'assurer la relecture technique. Raphaël Goetter, gourou des CSS, a également accepté de relire l'annexe B, en dépit de son planning de ministre. Le livre a énormément bénéficié de leurs apports et remarques constructives. Ce que vous y aimerez, vous le leur devrez certainement. Si certaines parties vous déçoivent, la faute sera mienne. J'adresse également toute ma gratitude à Tristan Nitot pour avoir accepté de rédiger la préface, ce que je considère comme un bel honneur.

Enfin, ma compagne Élodie Jaubert a supporté mon manque de disponibilité pendant les quelques trois mois d'écriture, et m'a soutenu sans faillir avec beaucoup d'amour, au point même d'accepter, au cœur de la tourmente, de devenir ma femme. Ce livre est là, avant tout, grâce à elle.

1

Pourquoi et comment relever le défi du Web 2.0 ?

Le Web 2.0, c'est bien, mais quels en sont les problématiques et les enjeux ? Que peut-on attendre à présent des sites, quelles technologies doit-on maîtriser (et peut-être apprendre à nouveau, apprendre mieux), et comment interopèrent-elles, notamment dans le cadre d'Ajax ? Ce chapitre tente de répondre à toutes ces questions, et termine en établissant un plan d'actions simple, basé sur cet ouvrage, pour vous aider à devenir un véritable expert des technologies Web 2.0.

Avant/après : quelques scénarios frappants

Afin de bien fixer les idées, explorons ensemble quelques services faisant un emploi efficace (et plutôt emblématique) des possibilités d'Ajax.

La saisie assistée : complétion automatique de texte

Une des principales utilisations novatrices d'Ajax est la saisie assistée, également appelée « complétion automatique de texte ». Le principe est simple, et courant dans les applications classiques : au fur et à mesure de la frappe, une série de valeurs finales possibles est proposée à l'utilisateur, qui correspond à ce qu'il ou elle a tapé jusqu'ici.

C'est un comportement qu'on attend d'un simple téléphone mobile (avec le fameux mode T9 pour la saisie des messages) tandis qu'il restait dramatiquement absent des pages web, lesquelles évoluent pourtant dans un environnement bien plus sophistiqué.

Voyons un premier exemple, avec www.ratp.info, le site de la RATP, la régie des transports en Île-de-France. Leur moteur de recherche d'itinéraires permet de préciser des adresses complètes, des stations ou des lieux comme points de départ et d'arrivée. En mode station par exemple, au fil de la frappe, une liste de possibilités est affichée qui permet de saisir rapidement la station visée.

Imaginons que l'on souhaite partir de la station « Arts et Métiers » à Paris. À peine a-t-on tapé `ar` que la liste (qui s'affiche par-dessus la saisie, ce qui est un choix discutable) présente l'aspect suivant :



Figure 1–1
Une saisie assistée sur le site de la RATP, après frappe de « ar »

Ajoutons simplement le `t`, et la liste devient :

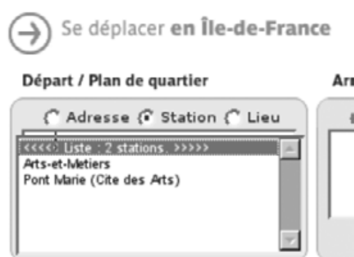


Figure 1–2
La saisie assistée après frappe de « art »

Il ne nous reste plus qu'à sélectionner directement la station (ce qui peut se faire en deux touches clavier ou d'un clic de souris), opération bien entendu plus rapide qu'une saisie complète et qui réduit dramatiquement le risque d'erreurs de frappe, améliorant ainsi la pertinence du moteur de recherche d'itinéraires.

Un exemple plus connu, et peut-être plus impressionnant, est Google Suggest, fonction proposée par les laboratoires de Google (http://labs.google.com, qui propose une foule de fonctions avancées encore en rodage).

Google Suggest est un mode spécial d'utilisation de la version anglophone du moteur de recherche : http://www.google.com/webhp?complete=1&hl=en. Voici ce qui se passe (instantanément ou presque !) lorsqu'on démarre une recherche sur Ajax en tapant le a initial :

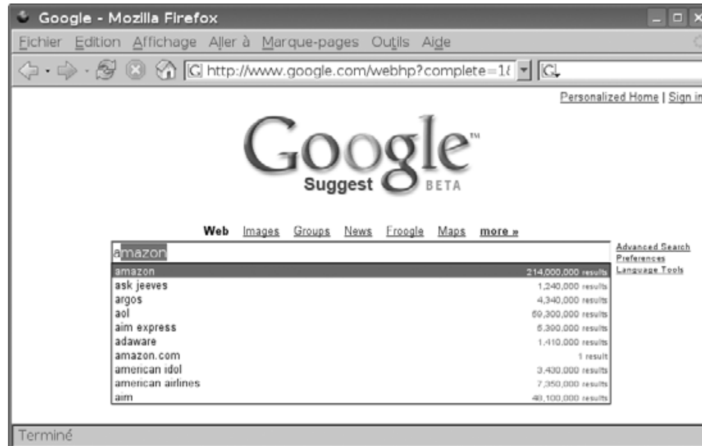


Figure 1–3
Saisie assistée avec Google Suggest, après frappe de « a »

La liste de propositions, qui est longue, inclut également le nombre de résultats, ce qui permet de rendre moins ambiguë la recherche (par exemple, en cas de doute dans l'orthographe de Johannesburg, la capitale de l'Afrique du Sud, à peine a-t-on tapé `johan` qu'on voit l'orthographe correcte obtenir près de 7 millions de résultats, tandis qu'en ajoutant immédiatement un `e`, on tombe sur des recherches ne dépassant pas les dizaines de milliers).

Voici l'affichage de Google Suggest alors que nous continuons notre recherche sur Ajax, en ayant tapé `aja` :

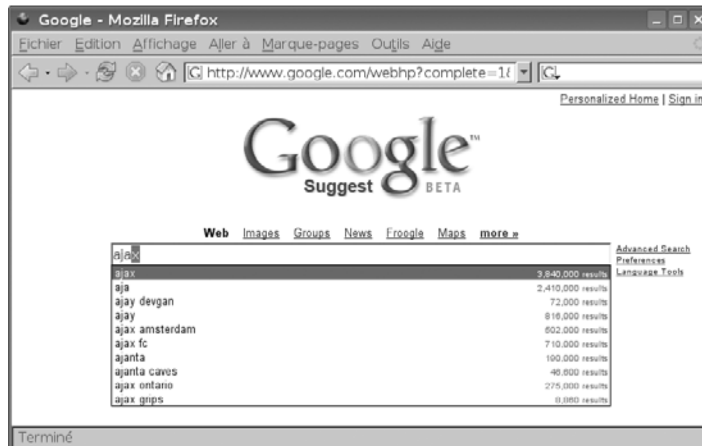


Figure 1–4
Saisie assistée avec Google Suggest, après frappe de « aja »

Notez que sur une connexion aujourd'hui classique (ADSL d'au moins 512 kbit/s), les requêtes en arrière-plan effectuées par le moteur de suggestion n'entravent pas le moins du monde le confort de frappe.

Le chargement à la volée

Une autre utilisation très courante d'Ajax réside dans le chargement de données à la volée. On peut d'ailleurs considérer la saisie assistée comme un cas particulier de chargement à la volée. D'une façon générale, l'idée reste l'obtention de contenu suite à une action utilisateur sans exiger le rechargement complet de la page.

Il peut s'agir de n'importe quel contenu : liste de propositions (comme nous l'avons vu précédemment), informations dynamiques (données météo, valeurs boursières), articles issus de flux RSS ou Atom (blogs, modifications apportées à un référentiel de sources, journaux d'information en ligne), graphiques (cartes, prises de vue par satellite, niveaux de jeu en ligne)… La seule limite reste l'imagination (et, dans une mesure chaque jour plus faible, la bande passante) !

Deux exemples incontournables donnent un aperçu des possibilités.

Tout d'abord Netvibes, jeune société française spécialisée dans les technologies Web 2.0, dont la page d'accueil fournit à tout un chacun un portail personnel en ligne entièrement personnalisable, tant dans le contenu que pour la disposition visuelle de ce contenu.

Voici la page d'accueil par défaut :

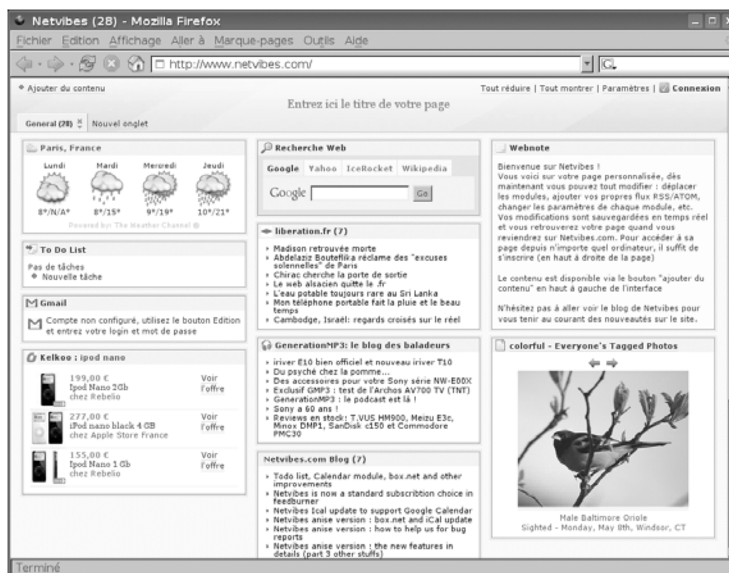


Figure 1–5
La page d'accueil par défaut de Netvibes

Notez la structure de la page :

- Une série de pavés, qui peuvent être déplacés librement et à volonté, par simple glisser-déplacer, afin d'arranger la disposition du contenu comme bon nous semble.
- Un titre personnalisable (il suffit de cliquer dessus pour le rendre éditable immédiatement).
- Des liens dans le haut permettant d'ajouter du contenu, de minimiser/restaurer les différents pavés, de régler quelques paramètres généraux (tout ceci sans recharger la page d'ensemble) et de se connecter à son compte Netvibes.
- Des onglets pour catégoriser le contenu afin d'éviter une page trop lourde visuellement.
- Pas de bouton de sauvegarde.

Quant aux contenus disponibles, ils sont de natures très diverses : météo, liste de choses à faire, consultation courriel (ici via Gmail), recherche de meilleurs prix, dictionnaires et encyclopédies, blogs, notes et même des catalogues de photos (ici via Flickr) ! Voici d'ailleurs l'interface d'ajout de contenu, qui apparaît à gauche de la page lorsqu'on active le lien correspondant :

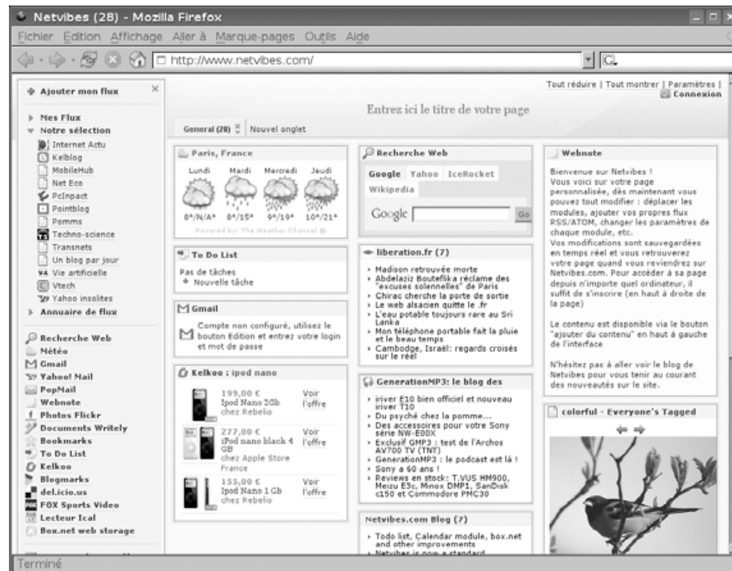


Figure 1–6
L'interface d'ajout de contenu de Netvibes

Chaque pavé dispose de propriétés spécifiques pour ajuster son comportement et son affichage, comme en témoignent les figures suivantes.

Figure 1–7
Modification des propriétés d'un pavé météo

Figure 1–8
Modification des propriétés d'un pavé de tâches à faire

Figure 1–9
Modification des propriétés d'un pavé de recherche des meilleurs prix

Chaque pavé peut être déplacé à l'aide d'un simple glisser-déplacer. Après avoir retiré quelques pavés, la figure 1–10 montre ce que l'on obtient tandis que l'on en déplace un.

Notez la délimitation en pointillés de l'emplacement cible, qui n'est peut-être pas très bien rendue à l'impression de cet ouvrage. Qu'à cela ne tienne : allez sur le site – qui ne nécessite aucune inscription – et essayez vous-même !

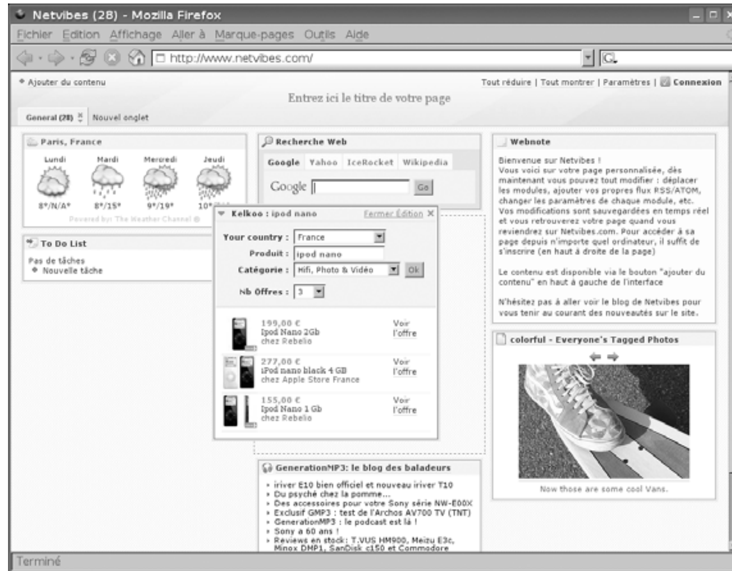


Figure 1–10
Déplacement d'un pavé par glisser-déplacer

Ces images permettent difficilement de rendre compte de l'impression que fait l'utilisation d'un service comme Netvibes. Pour dire les choses simplement, on a l'impression d'utiliser une application normale. En d'autres termes, la page ne souffre pas des limitations que nous associons habituellement au contenu web. Pas de rechargement global, très peu de délais, beaucoup de réactivité, une large place attribuée aux manipulations souris : ce ne sont là que quelques aspects qui, pour évidents qu'ils soient dans nos applications favorites, nous surprennent encore dans une page affichée par un navigateur. La seule zone d'ombre du site, qui tient dans une certaine tendance à la *divitis*[1], ne concerne en rien le confort de manipulation proposé à l'utilisateur.

Un autre exemple phare du chargement de contenu à la volée grâce à Ajax est le moteur de cartographie interactive Google Maps. Ce service en ligne permet d'effectuer des recherches géographiques (comme « hôtels à Paris » ou « 61 bld Saint-Germain 75005 Paris ») et de les associer à une cartographie détaillée, qui peut même être mélangée à des prises de vue par satellite. Il est également possible de demander des itinéraires.

Voici la vue initiale de Google Maps, recentrée sur l'Europe (figure 1–11).

1. http://fr.wikipedia.org/wiki/Divitis



Figure 1–11
Google Maps, vue centrée sur l'Europe, niveau de zoom faible

Par glisser-déplacer, on peut se déplacer dans la carte, aussi loin que l'on souhaite. On peut également faire glisser le curseur sur la gauche pour augmenter le niveau de zoom. Tout ceci suppose bien sûr que le contenu affiché est récupéré au fur et à mesure, sans quoi le volume de données à obtenir serait tellement énorme que toute tentative serait vouée à l'échec.

Le type de carte évolue suivant le niveau de zoom. Ainsi, en se concentrant sur Paris, on obtient une vue plus « routière » :



Figure 1–12
Google Maps, vue centrée sur Paris, niveau de zoom moyen

Et en cherchant une adresse (ou un itinéraire), on obtient une vue détaillée, avec en prime un ballon (notez l'ombre portée : on fait décidément dans la finesse pour l'interface utilisateur !) situant précisément l'emplacement concerné :

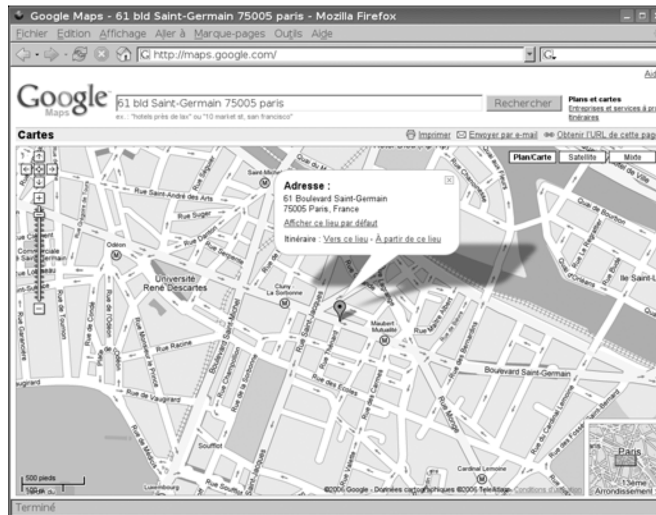


Figure 1–13
Google Maps, vue calé sur une adresse précise, zoom élevé

Google Maps permet également la reprise, et le mélange, des prises de vue par satellite qui font la base de Google Earth. Ainsi, pour ceux qui préfèrent du visuel à des cartes, ou pour ceux qui s'interrogent sur l'aspect de leur quartier, on peut passer en vue satellite ou mixte, et ce jusqu'à un niveau de détail impressionnant (1 pixel pour 50 cm) :



Figure 1–14
Google Maps, vue calée sur une adresse précise, en mode mixte, zoom maximal

Bienvenue sur le toit des éditions Eyrolles.

Le plaisir et le confort d'utilisation de ce service reposent sur l'interactivité forte qu'il permet : on utilise exclusivement la souris, on glisse, on zoome, on se déplace. La page ne se recharge jamais dans son intégralité, pas d'attente intermédiaire, pas de page blanche. Sans des technologies comme Ajax, un tel service serait un enfer ergonomique.

La sauvegarde automatique

Un aspect important de nombreuses interfaces web reposant sur Ajax est la sauvegarde automatique. En d'autres termes, on trouve de moins en moins de boutons Valider, Envoyer, Sauvegarder ou Enregistrer. Toute saisie est transmise automatiquement au serveur, en arrière-plan, une fois qu'elle est considérée terminée.

Si vous vous promenez à travers l'interface de Netvibes, par exemple, vous verrez qu'il n'y a pas de bouton dont le rôle est d'envoyer vos modifications au serveur. Et ce pour une raison bien simple : ces envois ont lieu de toute façon, en arrière-plan. C'est la base d'Ajax. Vous voulez changer le titre de la page ? Cliquez dessus, tapez le nouveau titre, validez avec la touche Entrée ou cliquez simplement ailleurs pour annuler la sélection du titre, et c'est tout.

On retrouve un schéma similaire dans un nombre grandissant de services d'achat en ligne, en particulier pour les hypermarchés sur le Web. Faire des courses est un processus plus exigeant, ergonomiquement, que les achats en ligne classiques. Dans ces derniers, on achète un nombre assez restreint d'éléments distincts : qu'il s'agisse de livres, CD-Rom, DVD, matériels de sport ou billets pour des spectacles, le panier reste relativement léger. Tandis que pour des courses, il enfle rapidement, pour atteindre plusieurs dizaines d'éléments distincts.

Devoir subir un aller-retour global pour chaque ajout découragerait l'internaute, et les hypermarchés l'ont bien compris. Toutefois, ils ont généralement recours à la technique éprouvée (et littéralement d'un autre siècle) des cadres (*frames*) afin d'aboutir à ce résultat : un cadre est dédié au panier, tandis qu'un ou plusieurs autres cadres affichent les rayons et produits.

Ce type d'architecture est certes bien connu et maîtrisé des professionnels du Web, mais si les principaux avocats des standards du Web (W3C, WHAT WG, WaSP) et de l'accessibilité dénigrent les cadres, ce n'est pas sans raison.

Outre les obstacles majeurs à l'accessibilité qu'ils présentent (ils complexifient grandement la navigation, en particulier au clavier, ainsi que pour les utilisateurs malvoyants et les logiciels qui les assistent), les cadres utilisés ainsi engendrent une complexité importante dans le code JavaScript nécessaire, sans parler des problèmes de compatibilité entre navigateurs.

De plus en plus de boutiques en ligne font le saut vers une architecture tout Ajax qui permet de simplifier radicalement le code côté client, tant au niveau JavaScript que HTML, en particulier en utilisant des bibliothèques telles que Prototype, ou des frameworks dédiés. Notons toutefois qu'un tel virage technologique doit être accompagné d'une politique efficace d'accessibilité, comme nous le verrons au chapitre 7.

Bien maîtriser ses outils clés : XHTML, CSS, JS, DOM et Ajax

Réaliser un site Web 2.0 ne repose pas sur une seule technologie. Comme on le verra au chapitre 5, Ajax lui-même n'est que la combinaison intelligente et novatrice de composantes techniques qui ne datent pourtant pas toutes d'hier.

C'est précisément cet existant qui peut jouer des tours au développeur, car il peut donner l'illusion d'une maîtrise déjà acquise, et l'amener à sauter, ou à tout le moins survoler, les chapitres et annexes dédiés à XHTML, CSS, JavaScript et le DOM. Or, traiter ces sujets à la hussarde, en se disant que « ça, je connais déjà », constituerait à mon humble avis une erreur.

Permettez-moi ici un bref aparté. Ces quatre dernières années, j'ai eu le plaisir d'enseigner à l'INSIA, école d'ingénieurs en informatique en alternance située à Paris. En charge de la spécialisation SIGL (systèmes d'information et génie logiciel), j'accorde une attention particulière aux retours d'expérience de nos étudiants (environ 700 depuis mon arrivée) sur leurs stages (3 jours par semaine en entreprise sur toute l'année scolaire, soient 10 mois), notamment lorsque ces stages impliquent du développement.

S'il est une impression qu'on retrouve très fréquemment dans leurs récits, c'est celle d'un faible niveau de compétences élémentaires en technologies web côté client (toutes celles dont traite cet ouvrage) chez leurs collègues, tant stagiaires que professionnels chevronnés. Mais si on leur demande d'argumenter leur sentiment, on retrouvera toujours les mêmes remarques : « il ne connaît rien au DOM », « elle n'a jamais entendu parler de balisage sémantique », « ils mettent des `div` partout, dont 95 % sont superflus », « ça ne marche que sur IE 6, et encore », « personne n'avait jamais ouvert une spécification du W3C, c'était totalement empirique », « à force de faire du Dreamweaver, ils étaient piégés quand ça ne marchait plus », etc.

Le constat fait réfléchir : il n'est pas rare de voir étudiants, jeunes diplômés et professionnels afficher la maîtrise ou l'expertise de ces technologies alors qu'en réalité, bon nombre d'entre eux ne maîtrisent que très partiellement ces sujets.

Essayer de déterminer les causes de ce décalage relève davantage de l'essai sociologique que du contexte de cet ouvrage. Mais il y a une leçon à en tirer, une conclusion décisive : les véritables experts sur ces technologies sont rares. Et comme tout ce qui est rare, ils sont précieux. Être le dépositaire de compétences précieuses, c'est bon pour sa carrière. Mieux encore, associer à ces compétences techniques un savoir-faire, une méthodologie de travail cohérente et efficace, c'est disposer d'un facteur différenciant de premier plan, d'un avantage compétitif indiscutable.

C'est précisément l'objectif de cet ouvrage : tenter de vous amener à ce niveau de compétences et de méthode. Aux questions suivantes, combien vous laissent indécis, voire vous sont incompréhensibles ?

- XHTML : Connaissez-vous la différence entre `abbr` et `acronym` ? Combien d'éléments `dd` sont possibles pour un même `dt` ? À quoi sert l'attribut `summary` de `table` ? Y a-t-il une différence entre les attributs `lang` et `xml:lang` ? À quoi sert `fieldset` ? Et `tabindex` ?
- CSS : À quelle version appartient `opacity` ? Comment fusionner les bordures des cellules d'un tableau ? Quelle est la différence entre `visibility` et `display` ? Peut-on appliquer une `margin` à un élément `inline` ? Pourquoi définir un `position: relative` sans chercher à repositionner l'élément lui-même ?
- JS : Qu'est-ce que l'*unobstrusive JavaScript* ? Que dire des attributs `href` commençant par `javascript:` ? Qu'est-ce qu'une fermeture lexicale ? Comment déclarer une méthode de classe ? Quels problèmes gravitent autour de la notion de *binding* ?
- DOM : À quoi sert la méthode `evaluate` ? Que se passe-t-il quand on insère un nœud déjà présent ailleurs dans le DOM ? Quel niveau a introduit les variantes `NS` ? Quelle différence y a-t-il entre nœud et élément ? Une `NodeList` est-elle utilisable comme un simple tableau ?

Ce livre n'apporte pas toutes les réponses à ces questions : s'il devait couvrir exhaustivement toutes ces technologies, vous tiendriez entre les mains un pavé de près de mille pages, sans doute cher et peu maniable. Mais vous trouverez tout de même de quoi acquérir des bases solides et, surtout, un sens de la qualité, qui vous permettront d'aller plus loin en toute confiance. D'ailleurs, la plupart des chapitres concluent par une série de ressources, papier ou en ligne, précisément dans cette optique.

Ne faites pas l'impasse sur les chapitres qui semblent déjà connus, déjà acquis. Si vous prenez le temps de les lire, j'aime à croire que vous y trouverez au moins quelques concepts ou données techniques qui vous étaient inconnus. Et puis, les sections finales de chaque chapitre sont là pour vous emmener encore plus loin.

Faire la part des choses : Ajax, c'est quoi au juste ?

Le terme « Ajax » est apparu pour la première fois dans un article de Jesse James Garret, sur le site de sa société Adaptive Path, le 18 février 2005 (http://www.adaptivepath.com/publications/essays/archives/000385.php). Il s'agit donc d'un terme relativement récent, qui est en réalité l'acronyme de *Asynchronous JavaScript + XML*.

L'article s'intitule Ajax: *A New Approach to Web Applications*. Il insiste sur le fait qu'Ajax n'est pas une technologie en tant que telle, mais la conjonction de technologies existantes pour une utilisation combinée novatrice. Ici comme ailleurs, le tout est pourtant bien plus grand que la somme des parties.

L'idée de base : la plupart des applications web, c'est-à-dire des applications dont l'interface graphique est affichée dans un navigateur, offrent une interaction pauvre et un confort d'utilisation plutôt restreint, en particulier si on les compare aux applications classiques, installées sur les postes utilisateurs. Dans une application web, on reste le plus souvent prisonnier du carcan requête/réponse : pour interagir avec l'application, qu'il s'agisse de valider la saisie de données ou de réorganiser les éléments d'une liste, il faut avancer à petits pas, avec à chaque étape un aller-retour entre notre navigateur et le serveur, qui engendre un rechargement complet de la page.

Par ailleurs, les possibilités offertes à l'utilisateur pour exprimer une demande ou réaliser une action restent primaires : il est rare de voir un site proposer d'ajouter un produit au panier simplement en glissant-déplaçant le premier sur le second. Ces limitations habituelles auraient très bien pu être levées sans utiliser Ajax, comme ce fut d'ailleurs le cas sur une minorité de sites. Toutefois, en bouleversant notre conception de ce qu'il était possible de proposer comme interactions sur une page web, Ajax a naturellement remis sur le devant de la scène la question des interactions riches, notamment au travers du glisser-déplacer et des effets visuels.

Ajax repose sur les technologies suivantes :

- XHTML pour assurer un balisage sémantique et cohérent du document, ce qui assure que celui-ci sera correctement représenté en mémoire et donc facilement manipulable.
- CSS pour habiller ce balisage sémantique et élargir la gamme des effets visuels utilisables pour communiquer avec l'utilisateur (comme signaler qu'un travail avec le serveur a lieu an arrière-plan, assister un glisser-déplacer, mettre en exergue un élément fraîchement ajouté à une liste).
- DOM pour représenter le document en mémoire, afin d'en manipuler la structure et le contenu sans avoir, justement, à recharger toute la page.

- XML et, dans une moindre mesure, XSL/T, pour structurer les données échangées en coulisses entre la page déjà chargée et le serveur, et transformer si nécessaire ces données en contenu affichable.
- `XMLHttpRequest`, service originellement fourni par Microsoft dans MSIE depuis sa version 5, jusqu'alors méconnu du grand public malgré un intérêt technique qui avait conduit à son implémentation ultérieure par les autres navigateurs. Le premier A de Ajax, c'est lui : le moyen de communication asynchrone.
- JavaScript enfin, qui relie tous ces éléments entre eux.

Là où une page web classique nécessite un rechargement à chaque action, aussi granulaire soit-elle (par exemple, la remontée d'un cran d'un élément dans une liste, afin de l'amener tout en haut de celle-ci), une page exploitant Ajax interagit avec le serveur en coulisses, sans se recharger entièrement. Un code JavaScript réagit à l'action de l'utilisateur en gérant un aller-retour interne avec le serveur, et met éventuellement à jour la page directement en manipulant le DOM de celle-ci. Le schéma suivant compare ces deux approches :

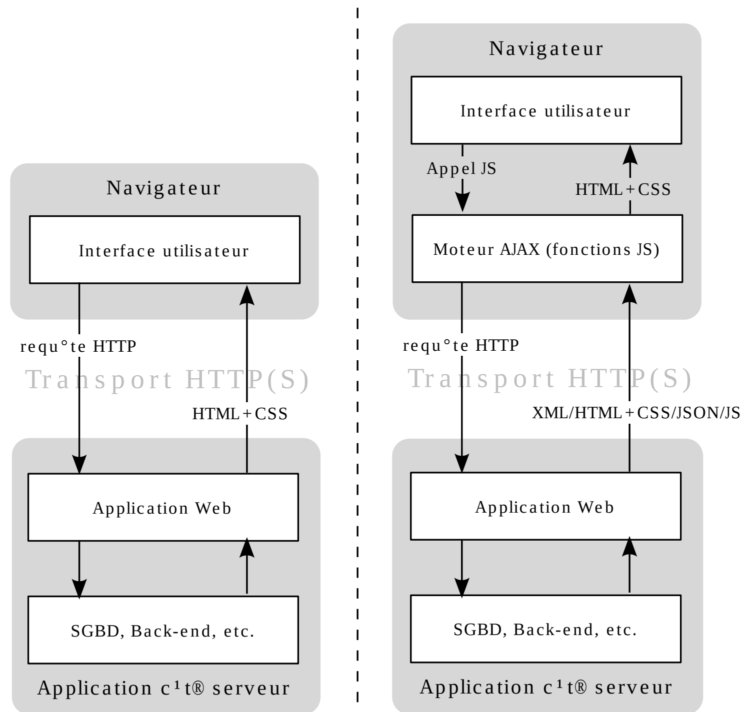


Figure 1–15
Approche traditionnelle et approche Ajax d'un écran d'application web

Vous aurez peut-être remarqué que dans la seconde approche, les données circulant du serveur vers le moteur Ajax sont marquées comme pouvant être non seulement du XML, mais aussi du contenu directement affichable (HTML+CSS), du JavaScript ou des données JSON (*JavaScript Object Notation*, http://www.json.org). En effet, rien ne contraint le type de données renvoyées : il appartient au développeur de déterminer ce qui semble le plus pragmatique, et le plus pratique, au cas par cas. Dans les exemples du chapitre 5, nous mettrons en œuvre plusieurs formats de retour pour illustrer quelques possibilités.

Ce qu'il faut bien comprendre, c'est la nature profondément asynchrone de la communication entre le moteur Ajax et le serveur : pendant que celle-ci a lieu, l'interface utilisateur est toujours présente, et surtout, toujours active. L'utilisateur peut continuer à utiliser la page, en effectuant d'autres actions tandis que la première est en cours de traitement.

Bien entendu, il peut être nécessaire de limiter les actions possibles pendant certains traitements. Par exemple, lorsqu'un produit est en cours de retrait d'une commande, l'utilisateur ne devrait pas avoir la possibilité d'en modifier la quantité. Permettre à l'utilisateur d'effectuer des actions sans attendre la complétion des précédentes impose la mise en place d'indications visuelles de traitement et de garde-fous. Nous verrons des exemples de mise en œuvre au fil des chapitres.

Plan d'actions pour deux objectifs : méthode et expertise

Pour tirer le maximum de profit de ce livre, voici un plan d'actions :

1. Commencez par l'annexe D pour configurer votre navigateur au mieux, afin de bien suivre les exemples de ce livre et d'augmenter radicalement votre productivité en développement web.
2. Si vous n'êtes pas très au point sur les fondamentaux « statiques » (XHTML, CSS), commencez par... les annexes ! Les annexes A et B auront tôt fait de vous (re)mettre en selle. Quant à aller chercher les informations de détail par la suite, rien de tel que l'annexe C pour vous apprendre à naviguer confortablement dans les spécifications. Par la suite, en lisant les chapitres, ne laissez pas une zone d'ombre vous irriter à l'arrière de votre lecture consciente : revenez aux annexes pour trouver l'information, ou y piocher les références de ressources détaillées aptes à vous la fournir. La différence entre le bon et l'excellent réside dans la maîtrise des détails autant que dans la vision globale.

3 À l'aise sur ces incontournables, prenez le temps de découvrir, ou redécouvrir, JavaScript et le DOM au travers de la première partie et des chapitres 2 à 4. Au-delà de la technique, vous y trouverez de très nombreux conseils et astuces méthodologiques, et la trame d'une démarche qualité pour votre code futur. N'hésitez pas à faire des exercices, à fouiller les documentations référencées, à faire de nombreux mini-projets ou simples pages de test. Seule la pratique mène à la perfection. Sans une véritable maîtrise de ces composantes, pas de bon développement Ajax possible !

4 Vous pouvez ensuite tout naturellement continuer sur Ajax à proprement parler, tant dans son aspect purement « communications asynchrones en arrière-plan » (chapitre 5) qu'au travers de frameworks établis permettant de donner un coup de fouet à votre productivité (chapitre 6) et vous ouvrant les portes d'effets visuels et de comportements impressionnants (chapitre 7). Bien employés, ces derniers donnent des interfaces haut de gamme. Ne laissez toutefois pas l'euphorie technique vous faire oublier l'importance des considérations d'ergonomie et d'accessibilité, qui font toute la différence entre le « peut mieux faire » et le « rien à redire ».

5 Pour ouvrir vos horizons et explorer des cas concrets d'utilisation, rien de tel que d'étudier des exemples d'interaction Ajax entre vos pages, éventuellement votre couche serveur, et les services disponibles sur le Web. Apprenez à combiner Ajax, services web, API REST et flux RSS/Atom pour obtenir des interfaces riches en contenu et en fonctionnalités.

Alors, prêts ? Partez !

Première Partie

Donner vie aux pages

Ça y est, on se lance. Vous choisissez de ne pas sauter cette partie, et vous avez bien raison. Même si vous pensez en connaître l'essentiel, voire toutes les ficelles, JavaScript et le DOM sont des technologies riches et complexes, dont on n'a généralement pas fait tout le tour, et qu'on a trop souvent découvertes de façon très empirique.

Pourtant, sans une véritable maîtrise de ces technologies, il est difficile de produire des sites web 2.0 de qualité, ou même simplement robustes et sans bogues.

Les deux prochains chapitres s'appliquent à vous fournir une connaissance solide et qualitative des bases de JavaScript et du DOM (d'aucuns trouveront d'ailleurs qu'ils vont bien plus loin que l'idée qu'ils se font des bases, ce qui donne une mesure de leur ampleur réelle !). Armés de ces connaissances fiables, vous n'aurez aucun mal à aller rechercher les détails supplémentaires dans les spécifications concernées (et si la forme de ces dernières vous rebute, un petit tour par l'annexe C vous remettra vite en selle).

Mais la productivité ne se satisfait pas toujours d'une connaissance parfaite des petits rouages internes : elle a souvent besoin d'outils plus évolués que le seul langage, les interfaces nues, etc. C'est pourquoi le chapitre 4 va faire vos délices avec l'extraordinaire bibliothèque JavaScript nommée Prototype, qui jouit déjà d'une énorme popularité. Grâce à elle, vous allez révolutionner votre façon d'écrire du JavaScript, et gagner en productivité de façon significative. En plus de cela, vous trouverez probablement l'écriture de code JavaScript plus... agréable et plus sympathique, peut-être même plus amusante.

Allez, on retrousse ses manches, et on va essayer de regarder JavaScript, ce langage si mal connu, voire mal compris, avec des yeux tout neufs...

2

Ne prenez pas JavaScript pour ce qu'il n'est pas

JavaScript est sans doute un des langages les plus incompris de la planète. Tout le monde pense le connaître, croit le maîtriser et le prend néanmoins pour un langage de seconde zone, aux possibilités limitées, même pas capable de faire de l'objet ou de gérer les exceptions !

Rien n'est plus faux. JavaScript est un langage dynamiquement typé doté de la plupart des possibilités usuelles, en dépit d'une syntaxe pas toujours très expressive. De nombreuses possibilités avancées, qui ouvrent les vannes à un véritable torrent de fonctionnalités puissantes, sont souvent mal connues voire inconnues des développeurs web. Et pourtant, le langage offre de quoi mettre sur pied des bibliothèques comme Prototype (étudiée au chapitre 4), qui rendent l'utilisation quotidienne presque aussi agréable que du scripting Ruby !

Ce chapitre est là pour rendre justice à JavaScript en détaillant certains points souvent mal connus et en faisant la lumière sur certaines fonctionnalités avancées qui restent trop souvent dans l'ombre. À l'issue de ce chapitre, vous serez plus à l'aise pour aller examiner le code de bibliothèques JavaScript avancées, comme Prototype ou script.aculo.us, et créer vos propres bibliothèques haut de gamme.

Mythes et rumeurs sur JavaScript

Commençons par battre en brèche certains mythes autour de JavaScript qui, comme tous les mythes, ont la vie dure.

JavaScript serait une version allégée de Java

Voilà une idée reçue très répandue, qui vient bien entendu de la similarité des noms entre les deux langages. Le créateur du langage l'avait d'abord baptisé LiveScript. Or, en 1995, Netscape sortait la version 2.0 de son navigateur : Netscape Communicator 2.0, qui continuait à pousser en avant Java, fraîchement mis au point par Sun Microsystems, au travers des applets. Netscape 2.0 fournissait également LiveScript, un langage de script censé rendre les pages plus vivantes, et qui permettait notamment de manipuler en partie les applets.

Dans un souci de marketing, le langage de script, dont on ne percevait pas encore l'extraordinaire potentiel, a été renommé JavaScript, et décrit comme un langage « complément de Java » dans un communiqué de presse commun de Netscape et Sun Microsystems : http://wp.netscape.com/newsref/pr/newsrelease67.html.

Néanmoins, les deux langages sont très différents. Java est un langage compilé (en code intermédiaire, certes, mais compilé tout de même), avec un système de typage statique, une syntaxe rigoureuse et assez verbeuse, et concentré sur la représentation de classes et d'objets.

JavaScript, en revanche, est avant tout un langage de script. Cela signifie qu'il est conçu pour être utilisé avec très peu de contraintes et une grande agilité : syntaxe minimaliste et plus flexible, typage dynamique, exécution interprétée, etc. Par ailleurs, en dépit de la présence de concepts objet (objets, instances, champs, méthodes, exceptions, etc.), les aspects plus avancés (héritage, polymorphisme, encapsulation, méthodes abstraites, interfaces, etc.) sont soit absents, soit pris en charge par une syntaxe confuse.

Ce n'est pas que Java est mieux que JavaScript, ou inversement : les deux langages répondent à des besoins radicalement séparés, sont conçus par des équipes parfaitement distinctes et suivent des évolutions tout à fait autonomes.

JavaScript ne serait basé sur aucun standard

Si JavaScript a vu le jour en tant que projet interne chez Netscape, il a été standardisé très tôt, dès sa version 1.1, par un comité de l'ECMA, organisme de standardisation international, qui continue de le faire évoluer. Le standard ECMA-262 spécifie EcmaScript, qui représente en quelque sorte le « JavaScript standard ». La seconde édition constitue également un standard ISO (ISO/IEC 16262 pour les curieux).

La troisième édition, qui est aussi la version actuellement finalisée, correspond à JavaScript 1.5 et date de décembre 1999. Tout le travail d'évolution de JavaScript a lieu dans le groupe de travail TG1 à l'ECMA, dirigé par l'inventeur original du langage, Brendan Eich, employé par la fondation Mozilla.

Une version intermédiaire 1.6 est actuellement prise en charge par Firefox 1.5, et une version 1.7, qui inclura déjà des améliorations radicales inspirées par d'autres langages de script (générateurs, itérateurs, compréhensions de tableaux, affectations multiples, etc.) est disponible dans Firefox 2, sorti à l'automne 2006.

JavaScript serait lent

À l'origine, JavaScript était lent, comme l'étaient Java, Perl, Python ou Ruby. Mais la technologie des langages de script et des machines virtuelles a énormément évolué depuis, et tous ces langages sont aujourd'hui relativement rapides. Dans la pratique, on n'a plus de difficulté à faire tourner rapidement des pages applicatives complexes mettant en œuvre de nombreux gestionnaires d'événements, des requêtes Ajax simultanées, et des mises à jour visuelles dues aux règles CSS.

On peut même sortir de la page web pour se pencher sur le navigateur lui-même : l'interface graphique de Firefox, par exemple, est décrite en XUL, un langage basé sur XML, et son fonctionnement est écrit en JavaScript. Même chose pour Thunderbird. Ce n'est d'ailleurs pas sans raison que les fichiers de préférences de ces programmes, nommés `prefs.js`, décrivent les préférences utilisateurs sous forme de code JavaScript. Et pourtant, ces deux programmes sont très confortables à l'utilisation !

JavaScript serait un langage jouet, peu puissant

Ses origines modestes ont enraciné l'image d'un langage jouet, image dont souffrent d'ailleurs presque toujours les langages de script, en particulier aux yeux des aficionados de langages plus rigoureux, plus « sérieux », comme C++ ou Java.

Et pourtant, il y a 10 ans, Java lui-même était dans le camp des « petits », des langages jouets. Davantage d'outils apparaissent chaque jour dans l'univers Linux, qui sont écrits en Python ou en Ruby. Et aujourd'hui, de plus en plus d'applications sont réalisées sur la base de XULRunner, la plate-forme d'exécution de Mozilla basée sur XUL, C++ et JavaScript.

Indépendamment de son utilisation commune, JavaScript n'en est pas moins un langage doté des quelques fonctionnalités critiques aptes à lui permettre de faire de grandes choses. Lesquelles, d'ailleurs, manquent souvent aux langages de meilleure stature. Dans ce chapitre, nous aurons l'occasion d'en découvrir un certain nombre ; attendez-vous à d'agréables surprises...

S'y retrouver entre JavaScript, EcmaScript, JScript et ActiveScript

La compréhension de JavaScript n'est pas facilitée par la pluralité du paysage. On entend parler de JavaScript, de JScript, d'ActiveScript, d'EcmaScript... Comment s'y retrouver ? Voici quelques points de repère.

- JavaScript est le langage d'origine, qui s'appelait LiveScript avant de faire sa sortie publique. Après sa version 1.1, il évolue au sein de l'ECMA, organisme de standardisation au niveau international, sous le nom coquet de ECMA-262, dont la 3e édition (version actuelle) correspond à JavaScript 1.5. La prochaine édition, due au deuxième trimestre 2007, constituera JavaScript 2. Ce langage est pris en charge, dans le respect du standard, par la vaste majorité des navigateurs, dont (naturellement) Mozilla, Firefox et Camino, mais aussi Opera, Safari et Konqueror. MSIE prend en charge l'essentiel (voir ci-dessous).
- JScript est le nom donné par Microsoft à son implémentation de JavaScript. Il s'agit grosso modo d'une prise en charge à 95 % de JavaScript, augmentée d'un certain nombre d'extensions propriétaires, par exemple la classe `ActiveXObject` et la méthode `GetObject`. Cette variante n'est, bien entendu, disponible que sur MSIE, et c'est elle qui est documentée dans le MSDN. MSIE 6 fournit Jscript 5.6, qui correspond à peu près à JavaScript 1.5.
- ActiveScript n'est pas un langage, mais le moteur d'exécution de scripts, sous Windows, qui permet à une application d'exécuter du code dans plusieurs langages de script, généralement mis à disposition du moteur sous forme de plug-ins.
  Ainsi, Windows 2000 et Windows XP, qui disposent d'un service nommé WSH (*Windows Scripting Host*), fournissent ActiveScript (c'est pourquoi en double-cliquant sur un fichier `.js` sous Windows, il tentera d'exécuter le script comme un programme classique ; WSH est une source majeure de failles de sécurité dans Windows...). ASP et ASP.NET utilisent aussi ActiveScript.

Tout ce que vous ne soupçonniez pas : les recoins du langage

Voici le cœur du chapitre. Cette section sert un double objectif :

1. Reprendre des aspects du langage souvent traités par-dessus la jambe, et qui sont donc mal maîtrisés, mal connus, entraînant des erreurs d'utilisation par inadvertance et des débogages difficiles.

2 Mettre en lumière des aspects souvent inconnus du langage, parfois avancés il est vrai, mais qui constituent les fondations sur lesquelles repose l'extensibilité du langage. Sans ces aspects, des bibliothèques aussi furieusement utiles que Prototype, par exemple, n'auraient jamais pu voir le jour.

Variables déclarées ou non déclarées ?

Vous savez probablement qu'en JavaScript, il est inutile de déclarer de variables. Cette simple affirmation n'est pourtant pas tout à fait exacte. Une variable non déclarée se comporte différemment d'une variable déclarée. Prenons l'exemple de code suivant :

Listing 2-1 Différence entre variables déclarées et non déclarées

```
var total = 0;
var factor = 5;
var result = 42;

function compute(base, factor) {
    result = base * factor;
    factor *= 2;
    var total = result + factor;
    return total;
} // compute

alert('compute(5, 4) = ' + compute(5, 4));
alert('total = ' + total + ' -- factor = ' + factor +
    ' -- result = ' + result);
```

Selon vous, que va afficher ce script ?

Le premier affichage est sans piège : `result` vaut d'abord 5 × 4 = 20, `factor` passe à 4 × 2 = 8, et `total` vaut 20 + 8 = 28. On obtient en effet 28.

À présent, le deuxième affichage utilise nos variables `total`, `factor` et `result`. Il s'agit bien sûr des variables déclarées en haut de script, puisque notre affichage est hors de la fonction `compute`, et n'a donc pas accès aux déclarations qui y figurent. On devrait donc voir s'afficher les résultats de nos affectations : 0, 5 et 42, respectivement. Et pourtant, stupeur : on obtient 0, 5 et 28 ! Que s'est-il passé ?

C'est bien simple : dans une fonction, utiliser une variable sans la déclarer revient à utiliser une variable globale, créée pour l'occasion si besoin. Je dis bien « variable », car les arguments d'une fonction ne sont pas assujettis à cette règle : vous voyez que la modification de `factor`, dans la fonction `compute`, ne touche pas à la variable `factor` déclarée plus haut. On ne modifie que la valeur locale de l'argument `factor` passé à la fonction.

Puisque dans la fonction, `result` n'a pas été déclarée, et qu'une variable externe `result` existe, c'est cette variable qui sera utilisée. En revanche, `total` étant ici déclarée (mot réservé `var`), on obtient une variable locale, et on ne touche pas à la variable externe `total`.

Je vous conseille donc de toujours déclarer vos variables, quitte à le faire à la volée quand la syntaxe le permet, comme pour un index de boucle :

```
for (var index = 0; index < elements.length; ++index)
```

En effet, cela garantit que vous ne touchez pas aux variables globales existantes, et que vous n'avez pas de « fuites », en créant des variables globales inutiles, qui stockent une information normalement interne à votre fonction. Le script suivant illustre bien cette infraction au principe d'encapsulation :

Listing 2-2 Un exemple de « fuite » de donnée par non-déclaration

```
function secretStuff() {
    // traitement confidentiel, avec dedans :
    privateKey = 0xDEADBEEF;
    // fin du traitement
}
secretStuff();
alert(privateKey);
```

Les cas où vos fonctions veulent effectivement manipuler des variables globales sont rares, pour la simple raison que les variables globales sont, dans un code de qualité, rarissimes. Et je ne vous parle pas des oublis de `var` dans des fonctions récursives, c'est un souvenir trop douloureux...

Ceci dit, quand vous en déclarez néanmoins, vous vous demandez peut-être quel intérêt il y aurait à utiliser « `var` » devant le nom, puisqu'on est de toutes façons au niveau global ? Il n'y a pas d'intérêt technique clair, mais cela améliorera la lisibilité. Prenez par exemple le script suivant :

```
MAX = 42;
total = 0.0;
count = 0;
```

Comment savoir de façon certaine quelles déclarations constituent des constantes et quelles autres constituent des variables ? Bien sûr, le respect d'une norme de nommage, qui demande généralement qu'on écrive les constantes en majuscules, nous suggère que `MAX` est une constante, et `total` et `count` des variables.

Mais on n'est pas pour autant à la merci d'un développeur peu rigoureux, qui aura juste appelé ses données ainsi sans trop réfléchir, et n'adhère pas à la norme. Cas encore plus fréquent, peut-être `MAX` a-t-elle été initialement conçue comme une constante, mais au fil du temps la conception a évolué, et aujourd'hui le script la modifie parfois, sans qu'on ait pris la peine de renommer l'identifiant.

Voilà pourquoi des mots réservés comme `const` et `var` sont toujours utiles. La version explicite du script ne laisse pas de place au doute, et nécessitera une mise à jour lorsque l'évolution du code voudrait rendre `MAX` modifiable :

```
const MAX = 42;
var total = 0.0;
var count = 0;
```

Comment, vous ne saviez pas que JavaScript avait le mot réservé `const` ? Vous voyez que ce chapitre va vous apprendre des choses... Hélas, je suis bien obligé de mettre un bémol : `const` est une extension à JavaScript 1.5 par Mozilla. Il figure normalement dans les versions 1.6, 1.7 et la prochaine version 2.0, mais pas dans 1.5 et son standard de base, ECMA-262 3e édition. Aussi, MSIE et Opera ne le prennent pas en charge.

Par conséquent, vous ne trouverez hélas pas de `const` dans les exemples à venir et l'archive des codes source pour cet ouvrage. Les constantes seront simplement au niveau global, en majuscules, sans `var`. C'est le mieux qu'on puisse faire pour être portable sur MSIE et Opera. En revanche, si vous garantissez le parc client (Mozilla, Firefox, Camino, Konqueror), n'hésitez pas !

Types de données

Certes, JavaScript ne type pas ses variables, arguments et constantes, ce qui ne manque pas d'offenser les partisans des langages statiquement typés, comme C++, Java ou C#. Ceux qui évoluent quotidiennement dans des langages dynamiquement typés, comme JavaScript, Ruby, Python ou Perl, rétorqueront plutôt « et alors ? ».

Que le type ne soit pas déclaré ne signifie absolument pas qu'il n'y a pas de type. En revanche, une variable peut changer de type au gré de ses affectations. Et lorsqu'on tentera d'utiliser une variable contrairement à son type actuel, JavaScript nous rappellera à l'ordre en produisant une erreur.

JavaScript dispose tout de même d'un certain nombre de types fondamentaux. Ces types sont plus riches que les descriptions renvoyées pour leurs valeurs par l'opérateur `typeof`, soit dit en passant. Voici un petit tableau récapitulatif pour JavaScript 1.5, disponible à peu près partout.

Tableau 2–1 Types de données de JavaScript et valeurs de typeof

Type	typeof	Description
`Array`	`object`	Tableau classique, à dimensions quelconques (ex. `[]`, `[1, 2, 3]`).
`Boolean`	`boolean`	Valeur booléenne : `true` ou `false`.
`Date`	`object`	Date et heure : `new Date(…)`.
`Error`	`object`	Erreur survenue à l'exécution, généralement capturée par un `catch`.
`Function`	`object`	Cas particulier : on parle ici d'objets fonctions, obtenus en faisant `new Function(...) {...}`. Si on passe à `typeof` une fonction directement (ex. `typeof Math.sqrt`, ou encore `typeof document.createElement`), on obtient logiquement `'function'`.
`Math`	(voir texte)	`Math` est un singleton (il n'existe qu'un seul objet `Math` à tout instant, toujours le même) jouant un rôle d'espace de noms : `typeof Math` renvoie donc `'object'`. Pour les objets prédéfinis (tous les autres dans cette liste), `typeof LeType` renvoie `'function'`, car le nom de l'objet est assimilé à son constructeur.
`Number`	`number`	Nombre (toujours flottant en JavaScript), équivalent `double` (double précision IEEE 754).
`Object`	`object`	Un objet quelconque, y compris ceux issus du DOM.
`RegExp`	`function`	Une expression rationnelle (type d'objet). Si vous ne savez pas de quoi il s'agit, ou les maîtrisez mal, je ne saurais trop vous recommander d'apprendre (voir bibliographie de fin de chapitre). Disponibles dans pratiquement tous les langages, les expressions rationnelles sont fabuleusement utiles pour le traitement avancé de textes.
`String`	`string`	Une chaîne de caractères (type d'objet).

Tout objet dispose de propriétés (ou champs, ou attributs : des données dans l'objet) et de méthodes (ou opérations, ou fonctions membres : des fonctions dans l'objet). On peut également simuler la notion de propriétés et fonctions statiques.

Chaque objet a notamment une propriété `prototype`, extrêmement utile, que nous aborderons plus en détail un peu plus loin dans ce chapitre, à la section *Améliorer les objets existants*.

Fonctions et valeurs disponibles partout

JavaScript propose un certain nombre de fonctions globales, accessibles de n'importe où. Après une rapide liste, je reviendrai sur certains points délicats souvent méconnus autour de certaines fonctions.

Tableau 2–2 Fonctions globales de JavaScript

Fonction	Description
`Array`, `Boolean`, `Date`…	Chaque objet prédéfini a une fonction constructeur associée, qui peut assurer certaines conversions appropriées suivant le cas. C'est pourquoi `typeof Array` et `typeof Date` renvoient `'function'`, d'ailleurs.
`decodeURI`	Symétrique de `encodeURI` (voir plus bas).
`decodeURIComponent`	Symétrique de `encodeURIComponent` (voir plus bas).
`encodeURI`	Encode un URI (une URL, pour simplifier) conformément aux règles d'encodage URL (hormis les minuscules et majuscules non accentuées, les chiffres et certains signes de ponctuation, tout caractère est transformé en séquence hexadécimale `%xx` voire `%uuuu` si on est en Unicode). Laisse toutefois le début de l'URL (avant le `?` qui marque le début des paramètres) intacte. Par exemple, `'bonjour marc & olivier !'` devient `'bonjour%20marc%20&%20olivier%20!'`.
`encodeURIComponent`	Encode un composant d'URI/URL : ne laisse donc aucun caractère spécial intact.
`eval`	Fonction très importante : elle permet d'exécuter un code JavaScript stocké dans une chaîne de caractères. Cette capacité du langage à s'auto-exécuter est critique, et permet d'avoir des types de réponse JavaScript ou plus spécifiquement JSON dans un contexte Ajax, par exemple. On peut tout imaginer avec cette possibilité : code automodifiant, code délégué, et j'en passe.
`isFinite`	Permet de déterminer si la valeur stockée dans un `Number` est finie (`true`) ou infinie (`false`). Plus pratique et plus performant que les tests manuels sur `Number.NEGATIVE_INFINITY`, `Number.POSITIVE_INFINITY` et `Number.NaN`, par exemple.
`isNaN`	Seule manière fiable de détecter qu'un `Number` ne contient pas une valeur numérique valide (par exemple, s'il a reçu le résultat d'une division de zéro par zéro). En effet, la constante NaN (*Not a Number*) n'est, par définition, égale à aucun `Number`, pas même à NaN lui-même. Le test `(x == NaN)` échouera toujours…
`parseFloat`	Convertit un texte en nombre flottant. Néanmoins, le fonctionnement de la conversion est souvent mal compris, comme nous le verrons plus bas.
`parseInt`	Convertit un texte en nombre entier. Ne pas préciser la base peut causer des soucis en traitant des représentations utilisant un remplissage à gauche par des zéros, sans parler du mécanisme de conversion qui, comme pour `parseFloat`, est souvent mal compris (voir plus bas).

On trouve également trois valeurs globales, constantes ou non, dites propriétés globales :

Tableau 2–3 Propriétés globales de JavaScript

Propriété	Description
`Infinity`	Variable initialement à `Number.POSITIVE_INFINITY`. Je ne vois pas pourquoi la modifier, aussi préférez les propriétés plus explicites de `Number` (infinité négative et positive).
`NaN`	Variable initialement à `Number.NaN`, qu'on ne modifie normalement jamais ; juste un raccourci.
`undefined`	Variable équivalente à la valeur primitive du même nom. Permet de simplifier des tests du genre `('undefined' == typeof x)` en `(undefined === x)` grâce à l'opérateur d'égalité stricte.

Les mystères de parseFloat et parseInt

Toujours indiquer la base, sinon...

Commençons par expliciter le deuxième argument, optionnel, de `parseInt` : l'argument nommé `radix`. Il s'agit de la base numérique pour la conversion. Les valeurs possibles sont :

- 0 (valeur par défaut) pour une « détection » de la base (voir plus bas).
- 2 à 36, qui utilisent les chiffres de 0 à 9 puis autant de lettres de l'alphabet que nécessaire, sans prêter attention à la casse. Ainsi, en base 16 (hexadécimale), on utilise 0-9 et A-F. En base 36, on va jusqu'à Z.

De nombreux développeurs web ne savent même pas que ce deuxième paramètre existe ou ne pensent pas à l'utiliser. Le problème est que, s'il n'est pas précisé, il prend la valeur zéro, et aboutit à une détection automatique de la base selon les premiers caractères du texte :

- Si le texte commence par 0x ou 0X, la suite est de l'hexadécimal.
- S'il commence juste par 0 (zéro), la suite est de l'octal (base 8).
- Sinon, c'est du décimal.

C'est le deuxième cas qui pose régulièrement un souci.

Imaginez par exemple que vous avez un formulaire avec une saisie manuelle de date au format, disons, `jj/mm/aaaa`. D'ailleurs, un tel texte peut vous être transmis autrement que par une saisie de formulaire... Toujours est-il que vous souhaitez en extraire le mois. La plupart du temps, on procède (à tort) ainsi :

```
var month = parseInt(text.substring(3, 5));
```

Suivant le cas, `text.substring(3, 5)` renverra `'01'`, `'02'`... `'10'`, `'11'` ou `'12'` (pour un numéro de mois valide, en tout cas !). Là-dessus, vous appelez `parseInt` avec simplement cette portion de texte. Pour un mois jusqu'à juillet inclus, ça marchera. Mais août et septembre passent mystérieusement à la trappe ! Vous récupérez alors zéro.

Eh oui ! Faute d'avoir précisé la base (10, dans notre cas), vous avez laissé `parseInt` la détecter, et le zéro initial l'a fait opter pour de l'octal, où seuls les chiffres 0 à 7 sont autorisés. Il convient donc d'être toujours explicite dans l'emploi de `parseInt`, ce qui signifie généralement la base 10 :

```
var month = parseInt(text.substring(3, 5), 10);
```

Une conversion plutôt laxiste

Passons à présent à un comportement méconnu commun aux deux fonctions. Tout d'abord, il faut savoir que les espaces préfixes et suffixes sont ignorés, ce qui est en soi pratique :

```
parseInt('25') == parseInt('25') == parseInt('25 ') == 25
```

Ce qui peut piéger le développeur est le comportement de ces fonctions en cas de caractère invalide. On a deux cas de figure : un texte invalide dès le premier caractère, et un texte dont le premier caractère est valide.

Dans le premier cas, on récupère automatiquement `NaN` :

```
var result = parseInt('dommage', 10);
alert(isNaN(result)); // true
```

Dans le second cas, seul le début du texte est utilisé, et le reste est silencieusement ignoré, ni vu, ni connu ! C'est en particulier un problème pour de la validation de saisie, par exemple. Beaucoup de développeurs écrivent un code du type :

```
var value = parseInt(field.value, 10);
// ou parseFloat(field.value), selon ce qu'on veut...
if (isNaN(value))
    // traitement de l'erreur
```

Hélas ! Ça ne suffit pas. Dans un tel contexte, le texte « 42dommage » passera sans problème : on récupérera juste 42. Pourtant, nous souhaitons généralement vérifier que **l'ensemble du texte est valide**, pas seulement le début !

La solution est relativement simple : il suffit de convertir à nouveau la valeur numérique en texte. Si le résultat est identique au texte d'origine, on est tranquille ! À un détail près toutefois : les espaces de début et de fin, qui ne seront plus là.

Si vous souhaitez effectivement les ignorer, il faudra d'abord les supprimer vous-même, pour avoir un texte de référence comparable à celui que vous obtiendrez par la conversion réciproque. Une telle suppression se fait facilement avec la méthode `replace` des objets `String` :. Voici donc une fonction générique de test pour valeurs numériques :

Listing 2-3 Une fonction simple de validation de valeur numérique

```
function isValidNumber(text, intsOnly) {
    text = text.replace(/^\s+|\s+$/g, '');
    var value = intsOnly ? parseInt(text, 10) : parseFloat(text);
    return String(value) === text;
} // isValidNumber
```

Si l'expression rationnelle employée vous ébranle, je vous encourage encore une fois à apprendre leur syntaxe (par exemple, en apprenant sur http://www.expreg.com/presentation.php, qui mis à part sa spécificité PHP, présente les syntaxes universelles). Voici tout de même une explication :

- `^\s+` signifie « un nombre quelconque positif d'espaces en début de texte »
- `|` signifie OU
- `\s+$` signifie « un nombre quelconque positif d'espaces en fin de texte »
- `g` est un drapeau indiquant « remplace toutes les occurrences ». Sans lui, si on retirait des espaces au début, on n'en supprimerait pas à la fin...

On remplace les portions qui correspondent à ce motif par un texte vide, ce qui dans la pratique élimine les espaces préfixes et suffixes.

Que donne `isValidNumber('42dommage')` ? `intsOnly` n'étant pas spécifié, il vaut `undefined`, ce qui correspond au booléen `false`, et notre opérateur ternaire (`?:`) utilise donc `parseFloat`, lequel renvoie juste `42`. La conversion inverse en texte donne naturellement `'42'`, qui n'est pas égal à `'42dommage'`, donc on retourne `false`.

Et pour un texte vraiment inutilisable ? Par exemple, `isValidNumber('truc', true)` ? Ici on a `intsOnly` à `true`, donc on appelle `parseInt('truc', 10)`, qui renvoie `NaN`. Converti en `String`, cela donne évidemment `'NaN'`, qui n'est pas égal à `'truc'`. On renvoie bien `false`.

Voilà de quoi éviter bien des cauchemars.

Rappels sur les structures de contrôle

Les grands classiques

Vous connaissez normalement les grandes structures de contrôle classiques, qui sont les mêmes en JavaScript qu'en C, C++, Java, PHP et bien d'autres. Voici leurs aspects généraux, qui ne changent strictement rien à vos habitudes :

Tableau 2–4 Aspects généraux des structures de contrôle usuelles

if/else	for	while	do/while	switch/case

`if (cond)`
`    ....;`
`else if (cond)`
`    ....;`
`else`
`    ....;`
	`for (init; cond; incr)`
	`{`
	`    ....`
	`}`
		`while (cond)`
		`{`
		`    ....`
		`}`
			`do {`
			`    ...`
			`} while (cond);`
				`switch (expr) {`
				`    case expr:`
				`        ...`
				`        break;`
				`    case expr2:`
				`        ...`
				`        break;`
				`    default:`
				`        ...`
				`}`

Petite différence entre le `for` JavaScript et le `for` C++/Java ceci dit : si vous déclarez une variable d'index à la volée : `for (var index = ...`, cette variable en JavaScript ne sera pas locale à la boucle : elle existera dans la même portée que la boucle elle-même.

Vous connaissez aussi probablement les traditionnelles instructions `break` et `continue`, également présentes dans les langages cités plus haut : `break` quitte la boucle courante, tandis que `continue` saute immédiatement au tour suivant (ou quitte la boucle si c'était son dernier tour).

Labélisation de boucles

En revanche, tous les langages ne permettent pas à `break` et `continue` d'être labélisées. Il existe en effet une syntaxe qui permet d'affecter un nom à une boucle. L'avantage est qu'on peut alors fournir ce nom à un `break` ou un `continue`, qui vont fonctionner au niveau de la boucle ainsi nommée, plutôt qu'au niveau de la boucle courante. On peut donc sortir de, ou court-circuiter, plusieurs niveaux de boucle imbriqués.

Voici un exemple de recherche dans un cube de données, qui permet de quitter les trois niveaux de boucle dès que l'élément cherché a été trouvé (ce qui est largement préférable au fait de laisser les trois boucles se dérouler !), sans avoir à gérer un booléen `found` supplémentaire.

Listing 2-4 Un exemple pertinent de break labélisée

```
function findAndShow(cube, value) {
    var coords = null;
outerLoop:
    for (var x = 0; x < cube.length; ++x)
        for (var y = 0; y < cube[x].length; ++y)
            for (var z = 0; z < cube[x][y].length; ++z)
                if (cube[x][y][z] == value) {
                    coords = [x, y, z];
break outerLoop;
                }
    alert(coords
       ? 'Trouvé en ' + coords.join(',')
       : 'Pas trouvé');
} // findAndShow
```

for...in

Une boucle JavaScript mal connue, ou mal comprise, est le `for…in`. Sa syntaxe est très simple :

```
for (variable in object)
    ....
```

Contrairement à une idée reçue, ou simplement à l'intuition, cette boucle ne permet pas (hélas !) d'itérer sur les éléments d'un tableau ! Non, son rôle est tout autre : elle itère sur les propriétés de l'objet.

On peut donc obtenir une liste des noms des propriétés d'un objet quelconque avec le code suivant, par exemple :

Listing 2-5 Énumération des méthodes d'un objet avec for...in

```
function showProperties(obj) {
    var props = [];
    for (var prop in obj)
        props.push(prop);
    alert(props.join(', '));
} // showProperties
```

On utilise ici un aspect peu connu de JavaScript, que nous reverrons plus tard : les opérateurs `[]` et `.` sont presque synonymes. Pour accéder à une propriété d'un objet `obj`, alors que le nom de la propriété est stocké dans une variable `prop`, on ne peut pas faire `obj.prop` (cela chercherait une propriété nommée « `prop` »), mais `obj[prop]` fonctionne !

Ainsi, le code suivant :

```
showProperties(['christophe', 'élodie', 'muriel']);
```

produira l'affichage suivant :

```
0, 1, 2
```

Mais... De quoi s'agit-il ? En fait, sur un tableau, les propriétés détectables sont les indices existants du tableau. On a ici trois éléments, donc les indices 0, 1 et 2. Cet exemple n'est pas très convaincant, et nous n'aurons pas plus de chance sur tous les objets natifs de JavaScript : nous n'aurons que les propriétés (ou méthodes) ajoutées par extension de leur prototype, donc faute de code préalable, on n'obtiendra rien.

Tentons plutôt un saut en avant vers le chapitre 3, consacré au DOM, et testons ceci :

```
showProperties(document.implementation);
```

Sur un navigateur conforme aux standards, on obtiendra :

```
hasFeature, createDocumentType, createDocument
```

Ah ! Voilà qui est plus sympathique. Ce sont bien en effet les trois méthodes prévues par l'interface `DOMImplementation`, proposées par l'objet `document.implementation`. Notez qu'on a obtenu des méthodes, pas de simples champs. Les méthodes constituent aussi des propriétés.

Simplifier l'accès répétitif à un objet avec with

Pour terminer, précisons une dernière structure de contrôle fort pratique dans certains cas, qu'on retrouve également en Delphi (bien que les deux langages soient sans aucun rapport !) : le `with`.

Ce mot réservé permet d'étendre la portée courante en lui ajoutant celle d'un objet précis. Qu'est-ce que la portée ? Disons qu'il s'agit de l'ensemble des endroits où l'interpréteur va rechercher un identifiant. Par exemple quand vous tapez :

```
alert(someName);
```

Où JavaScript va-t-il chercher `someName` ? Par défaut, comme beaucoup de langages, il suit le chemin classique : le bloc courant, son bloc parent, et ainsi de suite jusqu'à la fonction. S'il est dans une méthode, il cherchera alors dans l'objet conteneur. Puis dans les éventuels espaces de noms contenant l'objet. Et enfin au niveau global. S'il ne trouve rien, vous obtiendrez une erreur.

Étendre la portée en lui rajoutant le contexte d'un objet précis est très utile pour simplifier ce genre de code :

Listing 2-6 Un exemple typique de code qui bénéficierait d'un with...

```
field.style.border = '1px solid red';
field.style.margin = '1em 0';
field.style.backgroundColor = '#fdd';
field.style.fontSize = 'larger';
field.style.fontWeight = 'bold';
```

Alors qu'avec `with`, ça donne :

Listing 2-7 Le même avec un with judicieusement employé

```
with (field.style) {
    border = '1px solid red';
    margin = '1em 0';
    backgroundColor = '#fdd';
    fontSize = 'larger';
    fontWeight = 'bold';
}
```

Pratique, vous ne trouvez pas ?

Opérateurs méconnus

JavaScript dispose de nombreux opérateurs, dont les grands classiques bien sûr, mais aussi certains moins connus. Il gère également certains opérateurs classiques d'une façon particulière.

Retour rapide sur les grands classiques

JavaScript reprend la plupart des opérateurs classiques. On retrouve notamment :

Tableau 2–5 Opérateurs classiques également présents dans JavaScript

Catégorie	Opérateurs
Arithmétiques	+, -, *, /, %, ++, --, - unaire (inversion de signe)
Affectations	=, +=, -=, *=, /=, >>=, <<=, &=, |=, ^=
Bit à bit	&, |, ^, ~, <<, >>
Relationnels	==, !=, >, >=, <, <=
Logiques	&&, ||, !
Textuels	+ et += pour la concaténation
Accès aux membres	obj.propriété et obj["propriété"] (déjà évoqués plus haut)

Mais on a aussi quelques opérateurs moins courants, même s'ils existent dans quelques autres langages.

Opérateurs plus exotiques

Par exemple, l'opérateur `===` (sa négation étant `!==`) représente l'égalité stricte. En plus d'être égaux en valeur, les opérandes doivent avoir le même type. On le trouve aussi, par exemple, dans PHP. Ainsi, `'3' == 3`, mais `'3' !== 3`.

L'opérateur `in` permet de déterminer si son opérande de gauche est bien une propriété de l'objet opérande de droite. C'est très pratique pour éviter les cas délicats de méthodes plus générales. Ainsi, on a l'habitude de tester que l'objet global `document` a bien une méthode `createTextNode` en faisant simplement ceci :

```
if (document.createTextNode)
```

C'est correct, car si la méthode manque, l'expression vaut `undefined`, équivalent ainsi à `false`, et le test échoue. Mais quid d'un test sur une propriété booléenne, ou simplement pouvant valoir `null` ? Par exemple, considérez le test suivant :

```
if (someDOMNode.firstChild)
```

Ce test ne permet pas de savoir si `someDOMNode` a bien une propriété `firstChild` : il suffit que `someDOMNode` soit un élément vide pour qu'il ait bien cette propriété, mais qu'elle vaille `null`, faisant échouer le test.

L'opérateur `in` à la rescousse !

```
if ('firstChild' in someDOMNode)
```

Notez que l'opérande de gauche est une valeur (un nombre, un texte, etc.), qui correspond au nom de la propriété.

Comportements particuliers

En JavaScript, n'importe quelle valeur possède un équivalent booléen, pour pouvoir être utilisée dans le cadre d'une expression de test. La règle est simple : les valeurs `null`, `undefined`, `false`, `-0`, `+0`, `NaN` et `''` (chaîne vide) équivalent à `false`. Le reste équivaut à `true`.

Si les zéros signés vous étonnent, souvenez-vous que les nombres JavaScript sont des nombres flottants conformes au standard IEEE 754 : on gère le zéro positif et le zéro négatif. Quelle différence ? Un zéro positif est obtenu, par exemple, en divisant un

nombre par une infinité de mêmes signes, tandis qu'un zéro négatif est obtenu en divisant un nombre par une infinité de signes opposés.

C'était la minute culturelle, parce que franchement, le jour où vous aurez besoin de cette distinction dans vos scripts...

Il est important de savoir qu'il est donc possible de tout utiliser dans un tes et avec les opérateurs logiques. Il faut juste bien se souvenir quelles valeurs équivalent à `false`. Heureusement, la liste est intuitive.

Prise en charge des exceptions

Depuis sa version 1.4, JavaScript gère les exceptions, ce qui en améliore considérablement la robustesse. JScript a attendu sa version 5.6 (MSIE 6).

Rappelons rapidement qu'une exception, c'est une erreur d'exécution. Une condition exceptionnelle (car il ne devrait pas y avoir d'erreur, pardi !) dans laquelle le programme est soudain plongé. Expliquer dans le détail le bien-fondé des exceptions par rapport aux mécanismes plus anciens de signalement et de traitement des erreurs sort largement du cadre de ce chapitre. Si le sujet vous intéresse, consultez le site suivant : http://fr.wikipedia.org/wiki/Système_de_gestion_d'exceptions.

Les types d'exceptions prédéfinis

Bien qu'on puisse utiliser absolument toutes les expressions comme représentation de l'erreur, JavaScript définit tout de même, depuis la version 1.5, six types d'erreurs précis. Il s'agit de spécialisations de l'objet `Error`, lequel fournit principalement des propriétés `name` et `message`. La première fournit le nom de l'erreur, c'est-à-dire le nom de son type ; la seconde fournit le message passé au constructeur de l'objet.

Tableau 2–6 Types prédéfinis d'erreur en JavaScript 1.5

Type	Description
`EvalError`	Problème à l'exécution du code fourni à `eval`.
`RangeError`	Indique qu'un argument ou une variable avait une valeur hors de l'intervalle autorisé.
`ReferenceError`	Utilisation d'un identifiant inconnu (faute de frappe, etc.).
`SyntaxError`	Problème syntaxique à l'analyse du code fourni à `eval`.
`TypeError`	Indique qu'un argument ou une variable avait un type invalide. Par exemple, méthode inconnue.
`URIError`	Erreur de syntaxe dans un URI passé à `encodeURI` ou `decodeURI`.
`InternalError`	Erreur de l'interpréteur. Par exemple, récursion infinie détectée ! N'existe pas sur certains navigateurs...

Parmi ces types, on peut imaginer réutiliser `TypeError` et surtout `RangeError` dans nos propres scripts. Le reste exprime surtout des situations survenant dans les fonctions globales.

Capturer une exception : try/catch

Lorsque vous souhaitez capturer une exception susceptible de survenir dans un bloc de codes, vous devez encadrer ce bloc par un `try...catch`, selon le schéma suivant :

```
try {
    // code susceptible de lancer l'exception
} catch (e) {
    // traitement de l'exception, stockée dans l'objet e
}
```

Le principe, identique au traitement dans les autres langages gérant les exceptions, est le suivant : si le code encadré par `try` et `catch` ne lève aucune exception, le contenu du `catch` est ignoré, et l'exécution se poursuit sur la ligne suivant l'accolade fermante. En revanche, si une exception survient n'importe où dans le code encadré, le reste de ce code est court-circuité, et le `catch` est déclenché. L'exception est rendue accessible au travers de la variable nommée entre les parenthèses. Vous pouvez ainsi examiner son nom, son message et son type.

Dans la pratique, il est rare de voir survenir des exceptions issues du système, ou du navigateur lui-même. La plupart du temps, vous lancerez les exceptions vous-même, et les rattraperez plus haut dans la pile d'appels.

Voici tout de même un exemple d'exceptions natives, qui suppose que l'utilisateur a pu taper un code JavaScript dans un champ de saisie d'ID `jsCode`, et nous a demandé de l'exécuter (ce qui, hors d'une page d'exemples, constituerait un trou de sécurité béant !) :

Listing 2-8 Un exemple de traitement d'exceptions natives

```
function evalCode() {
    var code = document.getElementById('jsCode');
    var errorZone = document.getElementById('error').firstChild;
    errorZone.nodeValue = '';
    try {
        var result = eval(code.value);
        if (undefined !== result)
            alert('Résultat : ' + result);
    } catch (e) {
        if (e instanceof SyntaxError)
            errorZone.nodeValue = 'Erreur de syntaxe : ' + e.message;
```

```
        else if ('undefined' != typeof InternalError
                ➥ && e instanceof InternalError)
            errorZone.nodeValue = 'Erreur interne : ' + e.message;
        else
            errorZone.nodeValue = e.name + ' : ' + e.message;
    }
    code.focus();
} // evalCode
```

Observez l'utilisation de l'opérateur `instanceof`, qui permet de déterminer si l'objet en opérande gauche est compatible avec le type en opérande droit (s'il est de ce type ou d'un type descendant, donc). C'est ainsi qu'on distingue entre les différents types d'erreurs, en JavaScript. Afin que le code passe sur tous les navigateurs, on vérifie aussi que le type `InternalError` est bien reconnu.

Vous trouverez l'exemple complet dans l'archive des codes source pour ce livre, disponible sur le site des éditions Eyrolles. Si vous utilisez Konqueror, vous n'obtiendrez que les erreurs de syntaxe : Konqueror utilise une gestion particulière, à base de « rapports d'erreurs », pour les autres cas.

Voici quelques exemples d'affichage :

Tableau 2–7 Exemples de comportement de notre script

Code saisi	Résultat (Firefox 1.5)
`5 + 3`	Boîte de message « Résultat : 8 »
`x + 3`	`ReferenceError : x is not defined`
`5.times(3)`	**Erreur de syntaxe :** `missing ; before statement`
`(5).times(3)`	`TypeError : 5.times is not a function`
`function f() { f(); } f();`	**Erreur interne :** `too much recursion`

Voilà donc de quoi gérer aussi bien les erreurs natives que celles que nous pourrions signaler en lançant nous-mêmes une exception (ce que nous apprendrons à faire plus loin).

Extension : catch multiples et conditionnels

Les navigateurs Netscape 6+ et Mozilla (Mozilla, Firefox, Camino) proposent une extension à cette syntaxe pour JavaScript 1.5, que je signale par souci d'exhaustivité, mais qui doit être utilisée uniquement lorsque l'on possède la maîtrise de l'environnement client (pour un intranet avec des postes standardisés, par exemple).

L'extension permet d'obtenir une syntaxe similaire à celle du `try/catch` dans la plupart des langages, en ayant plusieurs blocs `catch`, afin de différencier plus clairement

les comportements, en général selon le type de l'exception. Voici ce que donnerait notre exemple précédent :

Listing 2-9 La fonction précédente, reformulée en catch multiples

```
function evalCode() {
    var code = document.getElementById('jsCode');
    var errorZone = document.getElementById('error').firstChild;
    errorZone.nodeValue = '';
    try {
        var result = eval(code.value);
        if (undefined !== result)
            alert('Résultat : ' + result);
    } catch (e if e instanceof SyntaxError) {
        errorZone.nodeValue = 'Erreur de syntaxe : ' + e.message;
    } catch (e if e instanceof InternalError) {
        errorZone.nodeValue = 'Erreur interne : ' + e.message;
    } catch (e) {
        errorZone.nodeValue = e.name + ' : ' + e.message;
    }
    code.focus();
} // evalCode
```

C'est sympathique, mais cela n'ajoute rien d'un point de vue fonctionnel : il s'agit juste de sucre syntaxique. Aussi, je vous conseille de vous abstenir pour des pages n'ayant pas la certitude d'être vues sous un navigateur Netscape 6+ ou Mozilla.

Point important : une clause `catch` ne peut capturer que les exceptions survenues dans le bloc `try`, pas dans un autre `catch` ! Celles-ci se propagent normalement.

Garantir un traitement : finally

Le `try/catch` est bien pratique, mais il lui manque une alternative. En effet, imaginons que nous souhaitions seulement capturer une partie des exceptions possibles (voire aucune), mais garantir qu'un traitement donné soit exécuté en fin de code dangereux.

Les exemples ne manquent pas, car ce besoin apparaît à partir du moment où on alloue une ressource temporairement : fichier, *socket*, mémoire, etc. On souhaite l'ouvrir ou l'allouer, la manipuler, et assurer sa fermeture ou libération, qu'on ait rencontré une erreur ou non.

Ce besoin fréquent est la raison d'être de la clause `finally`. Cette clause peut être ajoutée au `try`, après tous les éventuels `catch`. Il ne peut y en avoir qu'une. Un `try` a donc au moins un `catch` ou un `finally`. Le contenu de cette clause est exécuté quoi qu'il arrive.

On peut imaginer un code du genre :

```
var res = new HeavyResource(args);
try {
    // Code manipulant res, qui a potentiellement des erreurs
} finally {
    delete res;
}
```

Qu'il y ait des erreurs ou pas, on libère la ressource. C'est très pratique (et cela manque un peu à C++, par exemple, qui doit recourir à des astuces de *smart pointers* pour obtenir le même résultat).

Lancer sa propre exception : throw

Tout l'intérêt du mécanisme des exceptions est de nous permettre de signaler une erreur à un endroit du code qui n'est pas à même de la résoudre ; plus haut dans la pile d'appels, peut-être un endroit du code aura-t-il les informations nécessaires pour apporter un traitement satisfaisant, ce qu'il ne manquera pas de faire à l'aide d'un `try/catch` prévu pour capturer notre erreur.

Seulement voilà, comment la lance-t-on, notre erreur ? À l'aide de l'instruction `throw`. Celle-ci prend un argument unique, qui est l'objet représentant votre erreur. Il peut s'agir de n'importe quel objet, ce qui en JavaScript signifie n'importe quelle valeur : `42`, `'Dommage Éliane !'`, un objet à vous ou une instance d'erreur prédéfinie feront tout autant l'affaire.

Évidemment, si vous souhaitez pouvoir apporter un traitement générique des erreurs (pour maintenir un journal d'erreurs par exemple), il vaut mieux fournir à chaque fois des objets compatibles avec `Error` : des instances soit d'erreurs prédéfinies (voir tableau 2-6), soit d'objets qui vous appartiennent et qui sont extérieurement compatibles avec `Error` (ils ont donc des propriétés `name` et `message`). Voici un exemple :

Listing 2-10 Exemples de lancements d'exceptions

```
function CustomError(message, info) {
    this.name = 'CustomError';
    this.message = message;
    this.customInfo = info;
} // CustomError

try {
    // Code qui s'exécute
    throw new CustomError('souci !', 42);
    // Code court-circuité
```

```
} catch (e) {
    if (e instanceof CustomError)
        alert(e.message + ' / ' + e.customInfo);
    else
        throw e;
}
```

Comme on utilise ici une fonction constructeur, l'opérateur `instanceof` fonctionne correctement.

Vous voyez aussi une utilisation courante de `throw` : la repropagation d'une exception depuis un `catch`. En effet, un `catch` « tue » l'exception, qui ne se propage plus : elle est morte, pour ainsi dire. Il est toutefois possible de la relancer, puisque `throw` lance son argument comme exception. Une clause `catch` peut donc simplement « intercepter » une erreur, par exemple pour la consigner dans un journal d'erreurs.

Améliorer les objets existants

JavaScript est un langage à prototype (par opposition à un langage basé sur les classes et sous-classes pour réaliser l'héritage). Tout objet JavaScript est doté d'une propriété prototype, qui représente le modèle sur lequel l'objet en question se base. Un objet dispose automatiquement de toutes les propriétés de son prototype. JavaScript permet de modifier le prototype d'un objet à n'importe quel moment, et d'ajouter ou d'enlever des propriétés à tout instant.

Un peu de théorie sur les langages à prototypes

C'est une notion très déroutante pour les personnes habituées à un système de classes et d'instances, comme Delphi, C++, Java ou C#. Il est inutile d'en maîtriser tous les détails pour écrire la très vaste majorité des scripts, mais cela aide à comprendre le code de bibliothèques riches, comme celui de Prototype (justement).

Voici un petit résumé des principales différences, pour les curieux :

Tableau 2–8 Différences entre des langages à classes et à prototypes

Dans un langage à classes...	Dans un langage à prototypes...
On distingue les classes et leurs instances : les objets.	Tout est une instance.
On définit une classe avec une définition, on l'instancie avec une méthode constructeur.	On définit et on instancie des objets à l'aide d'une fonction constructeur (voir `CustomError` dans le listing 2-10).
On obtient une hiérarchie en référençant la classe mère dans la définition de la classe fille.	On obtient une hiérarchie en définissant un objet père comme prototype de la fonction constructeur fille.
L'héritage suit la chaîne de classes.	L'héritage suit la chaîne de prototypes.

Tableau 2–8 Différences entre des langages à classes et à prototypes (suite)

Dans un langage à classes…	Dans un langage à prototypes…
Les définitions d'une classe et de ses classes ancêtres fournissent l'ensemble complet et définitif des propriétés de toutes les instances de la classe.	Les fonctions constructeurs définissent un jeu initial de propriétés. On peut ajouter ou ôter des propriétés à tout moment pour un objet individuel, ou pour tous les objets basés sur un même prototype.

JavaScript est donc particulièrement flexible quant aux définitions des objets, ce qui permet d'obtenir certains idiomes très pratiques, comme les objets anonymes, ou la simulation d'héritage statique par recopie des propriétés d'un prototype dans un autre (au cœur de la bibliothèque Prototype actuelle).

Mais si tout ceci vous plonge dans la plus extrême confusion, soyez sans crainte : vous n'avez pas besoin de trouver le tableau 2-8 d'une limpidité cristalline pour comprendre ce qui va suivre.

Retenez simplement qu'en modifiant les propriétés (champs ou méthodes) du prototype d'une classe (techniquement, il faudrait écrire d'une fonction constructeur, mais quand je dis classe, avouez que cela vous paraît plus clair, non ?), on affecte toutes les instances de cette classe (tous les objets créés par cette fonction constructeur). Et il faut noter que l'on affecte aussi ceux créés avant la modification ! En effet, ils référencent tous le même prototype.

Si vous voulez réellement réfléchir en termes classiques (c'est le cas de le dire), vous pouvez voir cela comme des instances qui feraient toutes de la composition/délégation sur un singleton.

Mise en pratique

Pour ce qui nous intéresse ici, cette notion nous permet d'augmenter les possibilités d'objets prédéfinis. Vous venez peut-être de Java, et êtes tout dépité de ne trouver dans `String` ni `startsWith` ni `endsWith`, sans parler de `trim` ? Il vous manque des méthodes dans `Array` (ah bon ? Vous êtes sûr d'avoir bien regardé, parce qu'il y en a un paquet…) ? Qu'à cela ne tienne, rajoutez-les !

Nous allons voir ensemble deux exemples. Nous allons d'abord étendre `String`, en lui ajoutant les trois méthodes que nous venons d'évoquer (et même une quatrième, tant qu'à faire !). Ensuite, nous ajouterons à `Array` une des rares fonctionnalités qui lui manquent vraiment.

Étendre ses Strings

Voici comment étendre les possibilités de n'importe quelle valeur représentée par JavaScript à l'aide d'un objet `String` (y compris tout ce qui vient du DOM !).

Listing 2-11 Extensions sympathiques de String par son prototype

```
String.prototype.endsWith = function(suffix) {
    return this.length - suffix.length == this.lastIndexOf(suffix);
};

String.prototype.isBlank = function() {
    return null != this.match(/^\s*$/);
};

String.prototype.startsWith = function(prefix) {
    return 0 == this.indexOf(prefix);
};

String.prototype.trim = function() {
    return this.replace(/^\s+|\s+$/g, '');
};
```

Toutes les chaînes de caractères, peu importe leur provenance et leur date de création, disposent maintenant de quatre fonctions supplémentaires. Nous verrons un peu plus loin un exemple d'exécution.

Array et les injections

L'objet `Array` de JavaScript est sans doute l'un des plus méconnus. La plupart des gens se bornent à utiliser ses propriétés d'index (`data[0]`, etc.) et sa propriété `length`, pour boucler sur les éléments par exemple. Pourtant, en tant qu'objet, il dispose de nombreuses méthodes très pratiques, qui nous facilitent considérablement l 'écriture du code.

Rien que pour vous donner une idée, voici leur liste à partir de JavaScript 1.6, sans entrer dans les détails : `concat`, `every`, `filter`, `forEach`, `indexOf` (si !), `join`, `lastIndexOf`, `map`, `pop`, `push`, `reverse`, `shift`, `slice`, `some`, `sort`, `splice`, `toSource`, `toString`, `unshift` et `valueOf`. Je suis prêt à parier qu'au moins certaines vous sont inconnues, en particulier les méthodes d'itération : `every`, `filter`, `forEach`, `map` et `some`. Il faut avouer qu'elles sont apparues avec JavaScript 1.6, et MSIE ne supporte déjà pas tout le standard 1.5...

Toutefois, même sur un navigateur prenant en charge toutes ces méthodes (les navigateurs Mozilla typiquement), il manque une fonction courante des objets énumérables : l'injection.

Pour simplifier, un énumérable est une séquence d'éléments, sur laquelle il est possible d'itérer du premier élément jusqu'au dernier. Une injection sur un énumérable consiste à définir une valeur, dite accumulateur, avec une valeur initiale, et qui va évoluer en lui appliquant une fonction donnée pour chaque valeur de l'énumérable.

Par exemple, supposons un accumulateur de valeur initiale zéro, et une fonction d'injection qui, en recevant l'accumulateur et l'élément courant de l'itération, renvoie la somme des deux. En injectant cette fonction sur un énumérable de nombres, on obtient la somme. Un accumulateur démarrant à 1 (un) et une fonction renvoyant le produit, retournent le produit interne de l'énumération, etc.

C'est un mécanisme très utile, surtout si on ajoute une astuce : faute de valeur explicite pour la valeur initiale de l'accumulateur, on prend le premier élément de l'énumérable, et on itère à partir du second.

Voici le code JavaScript qui ajoute cette possibilité aux tableaux, en définissant tant qu'à faire deux utilisations communes de l'injection :

Listing 2-12 Ajout de l'injection aux tableaux, et de deux utilisations courantes

```
Array.prototype.inject = function(fx, acc) {
    if (0 == this.length)
        return acc;
    var start = undefined === acc ? 1 : 0;
    if (undefined === acc)
        acc = this[0];
    for (var index = start; index < this.length; ++index)
        acc = fx(acc, this[index]);
    return acc;
} // inject

Array.prototype.sum = function() {
    return this.inject(function(acc, num) {
        return acc + num;
    });
} // sum

Array.prototype.average = function() {
    return this.sum() / this.length;
} // average
```

Remarquez que le cas du tableau vide ne conduit à aucune erreur dans `average` : `sum` renverra `undefined`, `length` renverra zéro, et `undefined / 0` donne `NaN`, ce qui est plutôt bien pour la moyenne d'un tableau vide, qui n'a donc aucun élément pour produire une moyenne valide.

Nous venons de voir l'aspect théorique, mais que cela donne-t-il en pratique ? Dans l'archive des codes source pour ce livre, disponible sur le site des éditions Eyrolles, vous trouverez un exemple complet, dont le résultat est présenté à la figure 2–1.

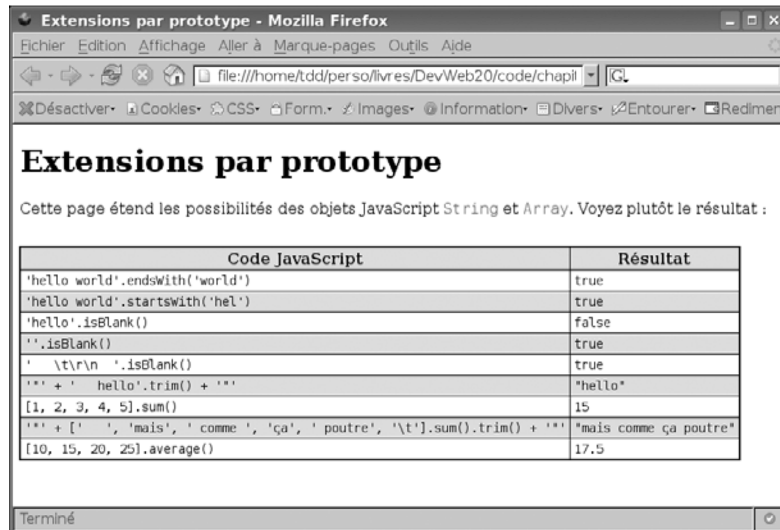


Figure 2–1
Exécution d'exemples utilisant ces extensions

Plutôt chouette, vous ne trouvez pas ? Les possibilités sont infinies... Enfin, pas tout à fait infinies quand même. Autant dans Mozilla, Firefox et Camino, tout objet a un prototype, autant d'autres navigateurs limitent cet aspect aux objets issus de fonctions constructeur. Ainsi, les objets du « DOM navigateur » (`window`, `navigator`, etc.) et les objets du DOM standard n'ont pas forcément un prototype. Le code suivant ne fonctionnera pas partout, hélas !

```
Element.prototype.myInnerText = function() {
    ...
}
```

Arguments des fonctions

JavaScript fait partie de ces langages qui n'exigent pas une concordance entre le nombre de paramètres et celui d'arguments. Qu'une fonction déclare ou non des paramètres (en fournissant leurs noms entre les parenthèses), il est possible de lui passer des arguments. Déclarer des paramètres ne sert qu'à donner un nom aux premiers arguments transmis. On peut en passer plus, moins, voire aucun.

Tout paramètre déclaré, pour lequel aucun argument n'est transmis, a la valeur spéciale `undefined`. Il existe deux moyens de tester cet état. Le plus long, et pourtant le plus répandu :

```
('undefined' == typeof paramName)
```

Le plus court, qui est d'aileurs le plus performant aussi, est moins répandu (par manque d'information vraisemblablement) :

```
(undefined === paramName)
```

Quant aux arguments passés en surplus, il faut y accéder d'une façon plus générique. En effet, chaque fonction a automatiquement une variable locale nommée `arguments`. Cette variable n'est pas à proprement parler un tableau (ce n'est pas toujours une instance de `Array`), mais elle se comporte comme tel, en fournissant une propriété `length` et des propriétés d'index, qui permettent donc la notation `arguments[numéro]`, en débutant bien entendu à zéro.

Voici par exemple une fonction qui affiche l'ensemble de ses arguments. Dans la mesure où `arguments` n'est pas forcément un `Array`, on va créer le texte à afficher manuellement puisqu'on n'a pas de garantie d'une méthode `join`. On verra toutefois à la prochaine section comment pouvoir utiliser `join` malgré cette incertitude.

Listing 2-13 Une fonction affichant la liste de ses arguments

```
function showArgs() {
    var msg = '';
    for (var index = 0; index < arguments.length; ++index)
        msg += arguments[index] + '\n';
    alert(msg);
} // showArgs

showArgs('hello', 1, 2, 3.14);
```

L'affichage obtenu donne ceci (la page d'exemple, fournie dans l'archive des codes source, illustre aussi la section suivante) :

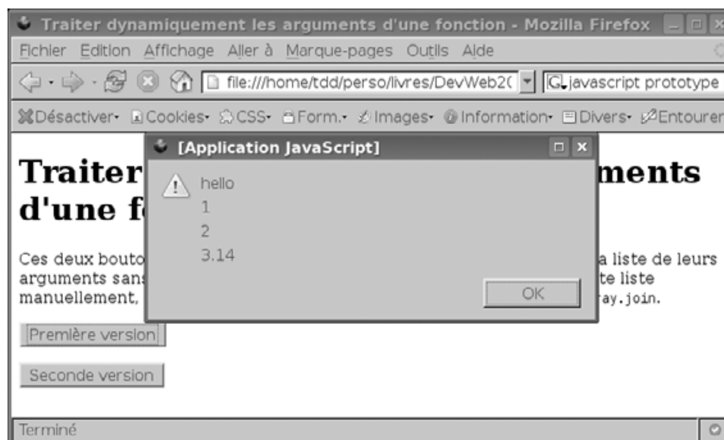


Figure 2–2
Notre fonction affiche bien ses arguments !

Cette variable `arguments` permet de réaliser facilement des fonctions à paramètres variables. Pour les habitués de C ou Java 5 : c'est comme la notation `...` de ces langages. Voici par exemple une fonction qui prend un facteur en argument, puis un nombre quelconque de valeurs, et renvoie le tableau des produits.

Listing 2-14 Un exemple de fonction à paramètres variables

```
function multiply(factor) {
    var result = [];
    for (var index = 1; index < arguments.length; ++index)
        result.push(factor * arguments[index]);
    return result;
} // multiply
```

Le binding des fonctions : mais qui est « this » ?

Voici l'un des sujets les plus délicats, et pourtant parmi les plus critiques, de JavaScript. Vous connaissez certainement le mot réservé `this`. Il s'agit d'une référence sur l'objet courant. Seulement voilà : quel est donc l'objet courant ?

Les règles ne sont pas très compliquées, mais on a vite fait de les oublier en pleine écriture de code, ce qui peut avoir des des conséquences désastreuses en termes de temps passé à déboguer...

- Au niveau global (code hors fonction, fonction globale), `this` représente l'objet `window` courant.
- Dans une fonction déclarée normalement et appelée directement, `this` représente l'objet sur lequel on a appelé la méthode.
- Dans une fonction anonyme, `this` a le même sens qu'immédiatement hors de la fonction anonyme. Si la fonction anonyme est affectée à une propriété d'un prototype, `this` désigne alors l'objet, fondé sur ce prototype, sur lequel on appelle la méthode.

La première règle est assez intuitive. Dans la seconde, il faut bien prendre garde aux contraintes : déclarée normalement elle implique l'utilisation de l'instruction `function` avec un nom de fonction, tandis que appelée directement elle signifie qu'on fait suivre le nom de la fonction par des parenthèses, au lieu de passer une référence à l'objet fonction pour exécution par un tiers.

Je suis bien conscient que pour les lecteurs, non habitués à fouiller dans les sémantiques d'appel et d'identité, ces quelques phrases sèment la confusion ! Essayons d'y voir plus clair avec quelques exemples.

Voici déjà un code sans surprise :

Listing 2-15 Exemples où this représente l'objet `window`

```
function f() {
    this.alert('Oui, this est bien window !');
}
alert(this.navigator.userAgent);
f();
```

Ce code fonctionne, car aussi bien dans la fonction globale `f` qu'au niveau supérieur, `this` représente l'objet `window`, propriétaire entre autres de la méthode `alert` et de l'objet `navigator`. D'ailleurs, tout code exécuté au niveau global est exécuté dans ce contexte, ce qui explique qu'on puisse appeler, par exemple, `alert`, sans le faire précéder de « `window.` » ou « `this.` ».

Voyons à présent le cas d'une méthode pour un objet donné, ici un objet anonyme :

Listing 2-16 Le sens normal de `this` **dans les méthodes d'un objet**

```
var myObj = {
    name: 'Mon objet à moi',
    method: function() {
        // Affiche le champ, car on est dans une méthode
        alert(this.name);
    }
}
myObj.method();
// Affichera autre chose, car ici this est l'objet window
alert(this.name);
```

Tout ceci semble parfait, mais que se passe-t-il dans le code suivant ?

Listing 2-17 Nous voilà trahis par le binding !

```
var myObj = {
    name: 'Mon objet à moi',
    method: function() {
        alert(this.name);
    }
}

function f(fx) {
    fx();
}

f(myObj.method);
```

Au premier abord, ce code semble devoir donner exactement le même résultat que le précédent : on passe notre méthode à `f`, qui l'appelle. Et pourtant, on obtient un affichage similaire (normalement, un texte vide) à l'exécution de `alert(this.name)` au niveau global, c'est-à-dire au niveau de `f` ou de la ligne finale. Que s'est-il passé ?

Il s'agit là d'un croque-mitaine fréquent chez les développeurs web, car la notion de *binding*, qui joue ici à plein, est rarement abordée par les ouvrages et didacticiels. Voici la clé du problème : lorsqu'on passe une méthode comme argument, on perd son *binding*. Plus exactement, elle prendra le *binding* de sa fonction appelante.

Trafalgar ! Que pouvons-nous donc faire ? Recourir aux méthodes méconnues des fonctions, pardi !

Voilà une locution qu'on ne risque pas de croiser tous les jours : « les méthodes des fonctions ». Souvenez-vous qu'en JavaScript, les fonctions sont des objets elles aussi : ce sont des instances de `Function`. Or, le prototype de `Function` prévoit deux méthodes spécialement conçues pour fournir un *binding* explicite à la fonction sur laquelle on l'appelle.

Ces deux méthodes s'appellent `call` et `apply`, et ne diffèrent que par la façon de passer les arguments à la fonction : `call` exige de les préciser un par un (ce qui suppose qu'on les connaît tous), tandis que `apply` utilise un tableau (ce qui permet des traitements génériques, `arguments` se comportant comme un tableau).

Pour illustrer ce point, nous allons réaliser trois petits exemples. Le premier corrige le tir sur notre fragment de code précédent, pour lui redonner le comportement attendu :

Listing 2-18 Notre dernier exemple avec le bon binding

```
var myObj = {
    name: 'Mon objet à moi',
    method: function() {
        alert(this.name);
    }
}

function f(fx, obj) {
    fx.call(obj);
}

f(myObj.method, myObj);
```

On passe ici le contexte (le *binding*) voulu à `f`, qui l'utilise comme premier (et unique) argument à `fx.call` (c'est-à-dire à `myObj.method.call`). Ce premier argument indique le contexte à utiliser par la méthode quand elle interprète `this`.

On peut même appeler call ou apply sur la fonction récupérée depuis le prototype. Ainsi, reprenons notre exemple de fonction affichant la liste de ses arguments, quels qu'ils soient. J'avais signalé tout à l'heure que comme arguments n'était pas un Array, on ne pouvait utiliser join dessus pour se simplifier la tâche. En effet, considérez le fragment suivant :

```
alert(arguments.join('\n'));
```

Il ne produirait qu'une erreur indiquant que join est introuvable. Or, il se trouve que join fait partie des méthodes dites « génériques » de Array, en ce sens qu'elles ont juste besoin que this ait une propriété length et des propriétés d'index, c'est-à-dire qu'on doit pouvoir faire this[0], par exemple. La méthode join ne modifie en rien l'objet sur lequel on l'invoque. Puisque arguments remplit ce contrat, il nous suffit d'invoquer join dans le contexte de arguments. N'ayant pas d'objet Array sous la main, on peut récupérer la méthode via le prototype :

Listing 2-19 Une version plus élégante de notre fonction showArgs

```
function showArgs() {
    alert(Array.prototype.join.call(arguments, '\n'));
} // showArgs
```

Ici encore, call est approprié, car on connaît le détail des arguments à passer à join : simplement '\n'. Lorsqu'on souhaite simplement passer les arguments d'origine par exemple, mais avec un contexte explicite, on préférera utiliser apply avec le arguments d'origine, qu'il soit modifié ou non.

Justement, pour revenir à notre exemple de méthode passée en argument, notre première solution possède le comportement voulu, mais c'est un peu laborieux : en plus de passer la méthode, on doit passer le contexte, et ceci à chaque descente dans une fonction, afin qu'au moment de l'utilisation effective de la méthode, on puisse faire un appel à call. Ce n'est pas pratique dans la majorité des cas. Comment pourrait-on passer une méthode en argument, sans perdre son contexte ?

Cette préoccupation est au cœur de l'extension bind présente dans la bibliothèque Prototype, qui a souvent besoin d'une telle possibilité. Elle n'utilise rien que nous n'ayons vu jusqu'à présent. À l'aide simplement d'une fonction anonyme et de apply, on peut obtenir ce résultat.

Listing 2-20 Binding durable d'une méthode pour passage comme argument

```
var myObj = {
    name: 'Mon objet à moi',
    method: function(extraArg) {
        alert(this.name + '\n' + extraArg);
    }
}

Function.prototype.bind = function(obj) {
    var fx = this;
    return function() {
        return fx.apply(obj, arguments);
    }
}

function f(fx) {
    fx(42);
}

f(myObj.method.bind(myObj));
```

Ici, un argument supplémentaire à `method` permet de voir qu'on ne perd pas les arguments au passage. L'appel final affichera bien « Mon objet à moi », et à la ligne, « 42 ». L'expression : `méthode.bind(objetVoulu)` renvoie une fonction anonyme, laquelle, lorsqu'on l'appelle, transmet tous ses arguments à la méthode d'origine, invoquée dans le contexte de `objetVoulu`, et renvoie le résultat. C'est une simple délégation.

Seule petite subtilité dans `bind` : la sauvegarde de `this` dans une variable locale avant de créer la fonction anonyme à renvoyer : dans celle-ci, `this` désignerait le `this` actif au moment de la définition de `bind`, soit l'objet `window`. Pas trop le tournis ?

Encore une fois, des exemples interactifs sont disponibles dans l'archive des codes source pour cet ouvrage, qui vous permettront de tester par vous-même les divers cas de figures évoqués.

Idiomes intéressants

On ne programme bien dans un langage que lorsqu'on commence à rédiger du code idiomatique, c'est-à-dire conforme aux usages du langage. À vouloir écrire du Java comme on faisait du C++, on ne produit que du mauvais Java. À écrire du Ruby sans se défaire de ses habitudes (qu'elles viennent de Perl, Java, C# ou Python), on écrit du mauvais Ruby. Il faut prendre le nouveau langage à bras le corps, et épouser ses concepts, ses formes et ses usages, pour produire quelque chose d'efficace, d'élégant et d'expressif.

JavaScript a traditionnellement été traité de manière assez superficielle, comme un langage de seconde zone dont les utilisations étaient trop simples pour mériter des idiomes de qualité. Et pourtant, dans les bibliothèques qui tirent vraiment partie de la puissance du langage, on trouve de nombreuses perles... Voici quelques bonnes idées extraites du code de bibliothèques comme Prototype ou script.aculo.us.

Initialisation et valeur par défaut avec ||

Commençons par voir comment fournir une valeur par défaut à un argument. On l'a vu, dans la mesure où il est parfaitement possible d'invoquer une fonction avec moins d'arguments qu'elle n'a de paramètres déclarés, certains paramètres vont se retrouver sans valeur.

Si ces paramètres ont ne serait-ce qu'une valeur valide qui soit compatible avec `false` (pour rappel, cela signifie `undefined`, `false`, `-0`, `+0`, `NaN` ou la chaîne de caractères vide, `''`), alors on n'a d'autre choix que de recourir à un test traditionnel, un rien verbeux :

```
if (undefined === arg)
    arg = expressionDeLaValeurParDéfaut;
```

En revanche, quand l'argument n'a comme valeurs valides que des expressions non compatibles avec `false` (un nombre positif, un élément du DOM ou son ID, etc.), alors on peut tirer parti de cette compatibilité en utilisant la sémantique de court-circuit de l'opérateur logique `||` :

```
arg = arg || expressionDeLaValeurParDéfaut
```

Cet opérateur fonctionne comme en Java, C, C++, C# ou Ruby, pour ne citer qu'eux : si son opérande gauche est équivalent à `true`, il le retourne, sinon il retourne celui de droite. Ici, si notre argument a une valeur valide, on la concerve, sinon on prend celle par défaut.

Si on n'utilise `arg` seulement à un endroit de la fonction, il est même possible de se passer de le redéfinir, et d'employer l'expression directement :

```
function doSomething(requiredArg, arg) {
    ...
    doSubTask(42, arg || expressionDeLaValeurParDéfaut, 'blah');
    ...
}
```

Sélectionner une propriété (donc une méthode) sur condition

Lorsqu'on souhaite choisir parmi deux propriétés d'un objet en fonction d'une condition simple, on a souvent recours, par souci de simplicité, à l'opérateur ternaire `?:`, comme ceci :

```
var value = someCondition ? obj.propA : obj.propB;
```

Dans tous les langages ayant des fonctions de premier ordre, les fonctions sont, par définition, manipulables comme des valeurs. On peut donc aussi s'en servir pour choisir une méthode :

```
var fx = someCondition ? obj.methodA : obj.methodB;
fx(arg1, arg2);
```

En JavaScript, on peut encore raccourcir cela en profitant du fait que l'opérateur `[]` est, comme on l'a vu, équivalent à l'opérateur de sélection de membre (`.`), basé sur le nom de la propriété. Par exemple, imaginons un objet ayant deux méthodes d'invocation identiques (elles prennent toutes les deux les mêmes arguments), et qu'on souhaite appeler l'une ou l'autre suivant une condition. On peut simplement écrire :

```
obj[someCondition ? 'methodA' : 'methodB'](arg1, arg2);
```

Exemple concret dans Prototype, tiré de `Element.toggle` :

```
Element[Element.visible(element) ? 'hide' : 'show'](element);
```

Encore mieux : appeler une méthode dont le nom est stocké, dynamiquement, dans une variable, mais dont on connaît les paramètres. Là aussi, voici un exemple concret tiré de Prototype (`Form.Element.getValue`) :

```
var parameter = Form.Element.Serializers[method](element);
```

Personnellement, je trouve cela franchement mignon...

Tester l'absence d'une propriété dans un objet

Nous avons déjà évoqué ce point plus haut avec l'opérateur `in`. Tester qu'un objet ne dispose pas d'une propriété (ce qui inclut les méthodes) ne peut pas toujours se résumer à ceci :

```
if (obj.propName)
```

Il suffit que l'objet dispose d'une propriété ainsi nommée, dont la valeur est équivalente à `false`, pour que le test ne serve à rien. Préférez l'opérateur `in`, qui sert expressément à ceci :

```
if ('propName' in obj)
```

Attention à bien encadrer le nom de la propriété par des guillemets simples ou doubles : l'opérateur gauche de `in` est un nom, donc un texte.

Fonctions anonymes : jamais new Function !

Nous avons déjà largement eu l'occasion de faire appel aux fonctions anonymes dans ce chapitre, par exemple dans la section sur l'extension des objets par leur prototype, ou sur le *binding*. Je souhaite juste préciser ici qu'il vaut mieux vous abstenir d'utiliser `new Function` dans ce genre de cas, et d'ailleurs, dans tous les cas.

Une fonction anonyme ainsi créée n'a pas le même *binding* pour `this` : au lieu d'utiliser le `this` du contexte environnant, elle utilise le `this` global, donc l'objet `window`. C'est très déroutant. Évitez cette syntaxe, préférez l'opérateur `function` qu'on utilise habituellement. Et c'est surtout plus court à écrire !

Objets anonymes comme hashes d'options

Nous avons eu l'occasion de voir des objets anonymes dans nos exemples de *binding*. Un objet anonyme est constitué d'une série de propriétés séparées par des virgules, le tout entre accolades. Chaque propriété (ce qui, je ne le répéterai jamais assez, inclut les méthodes) est définie par son nom, un deux-points (`:`) et sa valeur. Voici un exemple :

Listing 2-21 Un exemple d'objet anonyme

```
var author = {
    name: 'Christophe',
    age: 28,
    publisher: 'Eyrolles',

    greet: function(who) {
        alert('Bonjour ' + who + '.');
    },

    nickname: 'TDD'
}
```

On peut accéder aux propriétés avec l'opérateur point (.) Si le nom de la propriété enfreint la syntaxe JavaScript (par exemple, s'il s'agit d'un nombre), on utilisera l'opérateur `[]`. Un objet anonyme se comporte finalement comme un tableau associatif de valeurs, ce qu'on appelle communément un *hash*.

Cette similitude est très pratique lorsqu'on souhaite définir une fonction acceptant un certain nombre d'options, tout en évitant les signatures à rallonge. Une option, par définition, est optionnelle. Représenter une dizaine d'options sous forme d'arguments exigerait de passer toutes les options dans le bon ordre, etc. Ce qui n'est absolument pas agréable.

C'est pourquoi on accepte souvent les options sous forme d'un objet anonyme, dont les propriétés sont nommées d'après les options. Imaginons une fonction avec deux arguments obligatoires, offrant trois options `optA`, `optB` et `optC` ayant pour valeurs par défaut respectives `42`, la chaîne vide (`''`) et `false`. On suppose que pour `optA`, zéro n'est pas une valeur valide. Voici comment réaliser cela facilement :

Listing 2-22 Une fonction avec des options passées comme un hash

```
function myFunc(arg1, arg2, options) {
    // Dans le cas où on ne passe aucune option !
    options = options || {}
    options.optA = options.optA || 42;
    options.optB = options.optB || '';
    options.optC = options.optC || false;
    // Code de la fonction
} // myFunc
```

Évidemment, le code serait encore plus simple si JavaScript proposait un opérateur d'affectation combinée `||=`, mais ce n'est pas le cas. Pour appeler notre fonction, on dispose alors d'une syntaxe plutôt agréable, qui n'est pas sans rappeler les arguments nommés. Voici quelques exemples d'appel :

```
myFunc(1, 2);
myFunc(3, 4, { optB: 'Douglas Adams' });
myFunc(5, 6, { optA: 21, optC: true });
```

Simuler des espaces de noms

Lorsqu'on développe une grosse base de code, il est fréquent de vouloir la structurer. Suivant les langages, on dispose de mécanismes appelés paquets, modules, espaces de noms... En JavaScript, il n'y a rien (ce qui changera en JavaScript 2, soit dit en passant).

On peut toutefois simuler les espaces de noms en créant des objets anonymes ne jouant qu'un rôle de module. C'est tout simple :

```
var Ajax = {}
Ajax.Base = {
    // Définition de l'objet
}
Ajax.Updater = {
    // Définition de l'objet
}
// etc.
```

« Unobstrusive JavaScript » : bien associer code JS et page web

Il existe de multiples manières de faire en sorte que du code JavaScript soit exécuté dans une page. Du code peut s'exécuter au chargement du script (qui se produit bien avant que la page ne soit complètement chargée), par exemple pour initialiser une bibliothèque, ou en réponse à des événements.

Lorsqu'il s'agit d'associer du JavaScript à des événements, il est impératif que cela ne soit pas fait par le HTML. On ne met pas de JavaScript dans le fichier HTML pour les mêmes raisons qu'on n'y met pas de CSS :

1. Cela alourdit chaque page, et fait obstacle aux stratégies de cache des proxies et des navigateurs.
2. Cela constitue une intrusion du comportement (ou pour les CSS, de l'aspect) dans le contenu.
3. Les possibilités offertes par les attributs HTML sont beaucoup moins riches que celles offertes par le DOM pour l'association de fonctions JavaScript à des événements.

On évitera donc tout script intégré, du style de ceci :

```
<script type="text/javascript">
// du code ici
</script>
```

On évitera aussi tout attribut HTML événementiel, du style `<body onload="...">`. Le seul moyen acceptable de lier du JavaScript à une page consiste à placer ce JavaScript dans un fichier (généralement d'extension `.js`), et à charger ce fichier depuis la section `head` du HTML, comme ceci :

```
<head>
    …
    <script type="text/javascript" src="chemin/nom.js"></script>
    …
</head>
```

Au passage, un point important sur l'ancienne syntaxe utilisée pour préciser le langage du script :

```
<script language="javascript"…
```

Elle est dépréciée depuis HTML 4.01, donc depuis 1999, ce qui remonte déjà à plusieurs années. On utilise désormais l'attribut `type` avec le type MIME du langage, généralement `text/javascript`.

Idéalement, une page doit fonctionner sans aucun JavaScript. Les scripts doivent ajouter du confort, de la facilité, mais ne doivent pas constituer une condition sine qua non pour le fonctionnement des pages. Cela irait à l'encontre de l'accessibilité, et donc serait susceptible de gêner de très larges catégories d'utilisateurs.

Parce que la gestion portable, efficace et discrète (*unobstrusive*) des événements, et donc leur association à du code JavaScript, repose sur le DOM, vous verrez le détail de ces possibilités au chapitre suivant, dans la section « Répondre aux événements ». C'est aussi là que vous trouverez un traitement plus détaillé des relations entre JavaScript et accessibilité.

Je signale toutefois un cas où il est envisageable d'avoir une source XHTML avec un fragment de JavaScript à l'intérieur : la récupération d'un fragment XHTML par une requête Ajax. En effet, la couche serveur peut vouloir renvoyer aussi bien un contenu qu'une série d'opérations à effectuer côté client. Nous verrons des exemples de cela au chapitre 6, avec `Ajax.Updater` et son option `evalScripts`.

On peut parfois s'interroger sur le moment d'exécution d'un fragment de script : au moment de son chargement, ou une fois le chargement de la page terminé ? Vous trouverez un traitement détaillé de cette question au chapitre 3, à la section *Ne scripter qu'après que le DOM voulu est construit*.

Astuces pour l'écriture du code

En JavaScript, comme dans beaucoup de langages, on a intérêt à prendre quelques petites habitudes dans la rédaction de notre code source afin d'en améliorer la lisibilité et d'éviter certains pièges classiques. Voici quelques bonnes habitudes que je vous conseille.

Déjouer les pièges classiques

Connaissez-vous les termes *lvalue* et *rvalue* ? Ils désignent les deux opérandes d'une affectation : celui de gauche, qui constitue donc généralement une variable à laquelle on peut affecter une valeur, par exemple `index` ou `items[3].nodeValue` ; et celui de droite, qui constitue la valeur. Ce dernier est le plus souvent une expression ne constituant pas une variable, par exemple `42`, `3 * 7` ou `'hello' + name`.

Un piège fréquent dans de nombreux langages, dont JavaScript, consiste à faire une faute de frappe lors d'une comparaison de type `==`. On oublie un signe égal, et on obtient `=`, l'affectation. Au lieu de ceci :

```
if (x == 42)
```

on a alors cela :

```
if (x = 42)
```

Un tel code, qui est très suspect, ne déclenche qu'un avertissement en JavaScript. De tels avertissements ne sont pas immédiatement visibles sur la plupart des navigateurs. Même pour un développeur web équipé d'extensions dédiées, comme la Web Developer Toolbar de Chris Pederick, ou Firebug de Joe Hewitt, un avertissement cause seulement l'apparition d'une petite icône de panneau triangulaire jaune dans un recoin de l'interface : on ne s'en rend pas forcément compte.

Ce code affecte 42 à `x`, et renvoie `x`, qui vaut donc 42, ce qui est équivalent à `true`, et fait passer la condition. Avec une valeur fixe comme ceci, il est clair que c'est une erreur : si on sait qu'on y met 42, on sait que la condition sera toujours vraie, on n'aurait donc pas mis de `if`.

Pour éviter ce genre de problèmes, je vous recommande de toujours placer les *rvalues* à gauche dans une comparaison `==`. Prenons le code valide suivant :

```
if (42 == x)
```

Si vous oubliez par mégarde un signe égal, le code obtenu est le suivant :

```
if (42 = x)
```

Il ne fonctionne pas, tout simplement. Il génère une erreur, qui est souvent plus visible qu'un avertissement, et possède en outre l'immense avantage de stopper l'exécution du script, l'empêchant ainsi de continuer sur la base d'informations fausses, ce qui pourrait poser des problèmes. Imaginez qu'un tel code contrôle l'envoi d'un formulaire de suppression d'utilisateurs, et que le test vérifie qu'une case de confirmation est cochée… On causera certainement des dégâts si on opère sans de telles précautions.

Deuxième conseil : vous aurez peut-être remarqué que JavaScript n'exige pas toujours de terminer vos instructions par un point-virgule (`;`). Il s'agit en effet d'un séparateur d'instructions, et non d'un terminateur ; il est par ailleurs optionnel après les accolades fermantes. Hormis ce dernier cas, je vous conseille néanmoins de toujours utiliser le point-virgule à la fin d'une ligne de code : vous n'aurez pas à vous rappeler de le remettre le jour où vous rajouterez du code à la suite.

Dernier conseil, particulièrement destiné aux débutants : méfiez-vous d'un point-virgule tapé trop hâtivement derrière la parenthèse fermante d'un `if`, `for` ou `while` ! Par exemple, le code suivant :

Listing 2-23 Un script avec de gros problèmes…

```
if (42 < 21);
    alert('youpi !');
var msg = '';
for (var index = 0; index < 10; ++index);
    msg += index + ', ';
while (true);
    if (confirm('Quitter ?'))
        break;
```

Ce code affiche « youpi ! », laisse `msg` à la chaîne vide, et cause une boucle sans fin qui aura tôt fait d'épuiser l'interpréteur. Pourquoi ? Parce que le `if`, le `for` et le `while` ont un point-virgule en trop, discret, en fin de ligne, qui constitue une instruction vide. Comme, syntaxiquement, ces trois structures de contrôle ne gèrent qu'une instruction, c'est cette dernière vide qui est utilisée. En somme, le code précédent est équivalent au code suivant, où on repère bien mieux le problème :

Listing 2-24 Le même, reformaté

```
if (42 < 21)
    ;
alert('youpi !');
var msg = '';
for (var index = 0; index < 10; ++index)
    ;
```

```
msg += index + ', ';
while (true)
    ;
if (confirm('Quitter ?'))
    break;
```

Attention donc à ne pas être trop zélé avec les points-virgules...

Ces conseils valent pour tous les langages dont la syntaxe est proche, voire identique, sur ces points : C, C++, Java, C#...

Améliorer la lisibilité

Je recommande de toujours commenter la fin d'une fonction, au moins lorsque celle-ci dépasse les 10 lignes. On se retrouve très vite avec juste la fin d'une fonction visible, que ce soit parce qu'elle est très longue (ce qui indique un problème de modularité du code, soit dit en passant), ou parce qu'on a fait défiler le code source et que seule la fin subsiste. Dans les exemples de ce livre, la plupart des fonctions non triviales ont ainsi un commentaire de fin :

```
function compute(factor) {
    ...
} // compute
```

Après une condition ou une définition de boucle, allez systématiquement à la ligne et indentez, c'est tellement plus facile alors de voir quel code dépend de votre structure de contrôle. Évitez ce genre d'indentations impropres :

Listing 2-25 Une indentation très discutable

```
function badCode() {
    if (arguments.length == 0) throw new Error('Ooops');
    console.log('some log');
    for (var index = 0; index < 10; ++index) alert(index);
    console.info('some other info');
    while (document.body.hasChildNodes()) clearFirstNode();
    console.info('done.');
}
```

Préférez une indentation correcte :

Listing 2-26 Une indentation bien plus utile

```
function betterCode() {
    if (0 == arguments.length == 0)
        throw new Error('Missing arguments! Expected name.');
    console.log('some log');
    for (var index = 0; index < 10; ++index)
        alert(index);
    console.info('some other info');
    while (document.body.hasChildNodes())
        clearFirstNode();
    console.info('done.');
} // betterCode
```

Enfin, je ne saurai trop vous encourager à toujours adopter des conventions de nommage dans vos codes source, de préférence alignées sur les standards de l'industrie, ou en tout cas du langage employé. Voici quelques lignes de conduite simples, souvent adoptées dans d'autres langages répandus :

- Les types et espaces de noms utilisent une casse dite *CamelCase* majuscule, par exemple `Ajax`, `Uploader` ou `PeriodicalUpdater`.
- Les constantes utilisent une casse majuscule où les composants sont séparés par des soulignés (_) : `MAX`, `REGEX_SCRIPTS`.
- Les variables, arguments et propriétés utilisent une casse dite *CamelCase* minuscule, c'est-à-dire que la première initiale est minuscule. Quelques exemples d'expressions tout à fait réelles :
  `document.documentElement.firstChild.nodeValue`, `document.createElement`.
- Lorsqu'on crée des objets, je recommande chaudement d'utiliser une convention pour nommer les propriétés censées être privées (non accessibles depuis l'extérieur de l'objet) : préfixez-les par un tiret bas, par exemple :
  `_id`, `_each`, `_observeAndCache()`.
- Les noms de méthodes devraient toujours commencer par un verbe à l'infinitif, par exemple `createElement`, `removeChild` ou `adoptNode`. Les méthodes renvoyant un booléen devraient utiliser un verbe à la troisième personne du singulier, généralement au présent de l'indicatif, adapté à la sémantique de la méthode. Exemples : `hasChildNodes()`, `isBlank()`, `canClose()`.

Rien qu'en observant ces quelques règles simples, on améliorera sensiblement le code.

Mieux déboguer du JavaScript

JavaScript, c'est comme tout : plus nos outils sont bons, et plus on développe vite. Croyez-le ou non, encore aujourd'hui, la majorité des développeurs web n'utilisent pas d'outils évolués dans leur navigateur pour mettre au point leur JavaScript. Ces mêmes développeurs, qui frémissent d'horreur à l'idée de devoir développer en Java, C++ ou C# sans un EDI haut de gamme (et ne parlons même pas de Visual Basic), persistent à n'utiliser, pour travailler avec JavaScript, que l'équivalent d'un silex et d'un bout de bois. L'annexe D et cette section visent à vous élever au-dessus de cet état préhistorique.

Et pourtant, l'univers JavaScript propose les mêmes richesses que ceux des autres langages : éditeurs, complétion automatique, véritables EDI même, frameworks de tests unitaires et fonctionnels, extraction automatique de documentation, et j'en passe.

Il ne s'agit pas ici de passer tout cela en revue, mais de nous pencher plus particulièrement sur la question du débogage. Pour citer un membre du projet Mozilla, « déboguer du JavaScript, c'est un peu faire la chasse aux fantômes ». Mais il ne s'agit pas là d'une fatalité. Des outils peuvent vous aider considérablement, et nous allons en voir quelques-uns ici.

Avant de commencer, , j'affirme haut et fort que pour déboguer du JavaScript, rien ne vaut Mozilla Firefox. Ce navigateur figure déjà sur tous les postes de développeurs web un tant soit peu consciencieux, dans la mesure où, d'une part, il respecte infiniment mieux les standards que MSIE et, d'autre part, il est bien plus agréable à utiliser. Par ailleurs, il est disponible sur toutes les plates-formes, et représente plus de 20 % du marché mondial des navigateurs web, ce qui équivaut à plusieurs centaines de millions d'internautes.

Mais ce qui rend Firefox si utile pour les développeurs web, c'est son mécanisme d'extensions. Tout un chacun peut fournir des fonctionnalités supplémentaires, et cela autrement plus facilement qu'avec le mécanisme BHO (*Browser Helper Object*) disponible dans MSIE, qui est par ailleurs l'une des principales sources de problèmes de sécurité sous Windows. Lorsqu'on développe une page web, la meilleure méthode aujourd'hui consiste certainement à travailler d'abord sous Firefox pour mettre au point le HTML, les CSS et les scripts, puis à tester exhaustivement sur les principaux autres navigateurs, dont bien sûr MSIE, Safari et Opera. On sait en effet combien MSIE diffère des standards en termes de CSS et aussi (quoique dans une bien moindre mesure) de JavaScript.

La console JavaScript

Plusieurs navigateurs proposent une console JavaScript. Il s'agit d'une fenêtre qui recense les notifications de l'interpréteur JavaScript : erreurs, avertissements et messages d'information éventuels.

Certains navigateurs fournissent d'ailleurs un accès JavaScript à cette console, au travers d'un objet global dédié, ce qui permet à vos scripts d'y placer un message plutôt que de recourir au traditionnel mais pénible appel à `alert` (lequel vous propulse vite dans un tourbillon de clics).

Dans Mozilla Firefox

Dans Mozilla Firefox, la console est accessible via le menu Outils>Console JavaScript. La console ressemble à ceci :

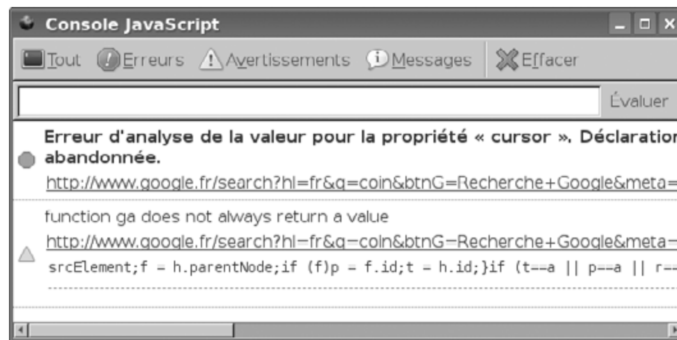


Figure 2–3
La console JavaScript de Mozilla Firefox

Notez qu'elle reçoit aussi des notifications de l'analyseur CSS (la plupart des messages d'erreur qu'on y rencontre sont souvent dus à des *hacks* CSS) et qu'elle conserve les notifications reçues par le passé : elle ne s'efface que manuellement, en cliquant sur le bouton Effacer de sa barre d'outils. Les quatre premiers boutons permettent de filtrer l'affichage suivant le type de message.

Par ailleurs il est possible d'y taper un fragment de code pour évaluation immédiate (mais sur une seule ligne), à l'aide de la zone de saisie en face du bouton Évaluer :

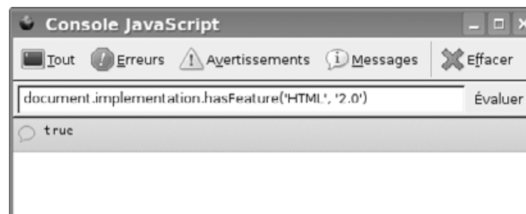


Figure 2–4
Évaluation directe dans la console JavaScript

Dans Opera

Opera fournit également une excellente console JavaScript, qui est en fait incluse dans la console d'erreurs, accessible via Outils>Avancé>Console d'erreur. Cette console récupère les erreurs de tous bords (comme CSS, JavaScript, SVG, HTML), et fournit de nombreux détails sur chacune. Des options de filtrage en bas à gauche de la liste des erreurs permettent de ne voir, par exemple, que les erreurs JavaScript.

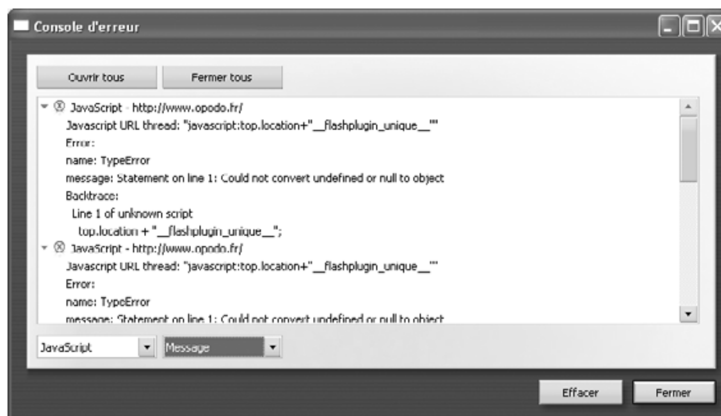


Figure 2–5
La console d'erreurs d'Opera, ici filtrée sur JavaScript

Vous trouverez de nombreux détails sur le débogage de JavaScript dans Opera à l'adresse http://my.opera.com/community/dev/jsdebug/.

Dans Safari

Safari fournit à partir de la version 1.3 un menu caché nommé `Debug`, qu'on peut activer en tapant dans un terminal (quand Safari n'est pas lancé) :

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

L'astuce figure dans la FAQ de Safari sur le site d'Apple, à la question 14 : http://developer.apple.com/internet/safari/faq.html#anchor14.

Une fois le menu `Debug` disponible, on y trouve une pléthore d'options, dont Show JavaScript console (Maj+Cmd+J).

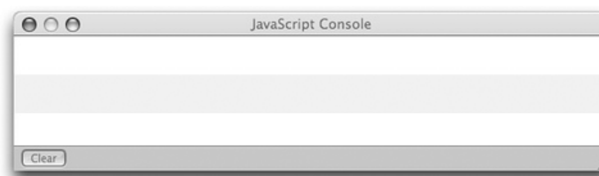


Figure 2–6
La console JavaScript « cachée » de Safari

Dans Konqueror

Konqueror propose également une console JavaScript, au sein de son débogueur JavaScript rudimentaire. Celui-ci n'est pas activé par défaut : il faut aller dans Configuration>Configurer Konqueror>Java & JavaScript>JavaScript et cocher les options Activer le débogueur et Rapports d'erreur. Le débogueur est alors accessible depuis le menu Affichage>Débogueur JavaScript (sur une page web, pas sur la page d'accueil spéciale de Konqueror).

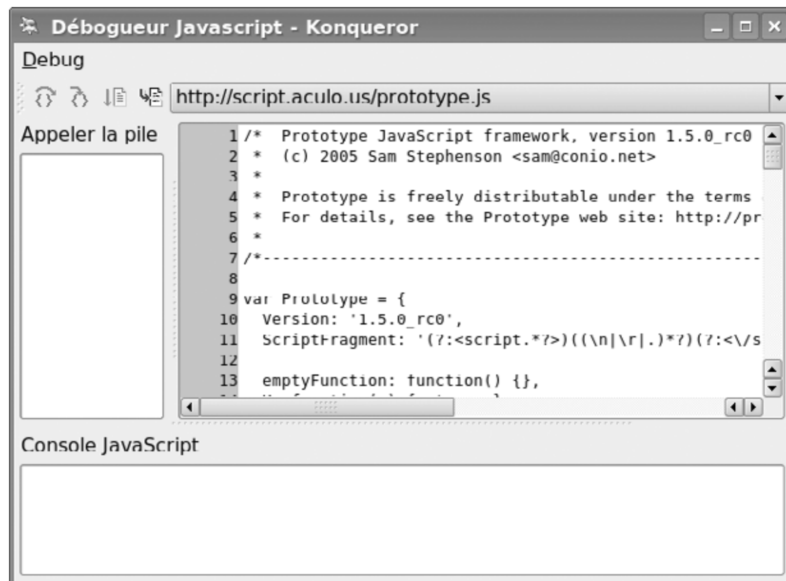


Figure 2–7
La console JavaScript dans le débogueur rudimentaire de Konqueror

Dans MSIE

MSIE 6 ne fournit pas de console JavaScript. Le mieux que l'on puisse faire consiste à cocher l'option permettant de voir une boîte de dialogue à chaque erreur de script, afin d'avoir quelques détails. Mais c'est pénible, et c'est quoi qu'il en soit une bien maigre consolation. Si vous souhaitez toutefois y recourir, l'option est dans Outils>Options Internet>Avancé>Navigation>Afficher une notification de chaque erreur de script. Il existe tout de même un débogueur téléchargeable à part : consultez l'annexe D pour plus de détails.

Venkman, le débogueur JavaScript

Je n'entrerai pas dans les détails des différents débogueurs JavaScript, mais nous allons examiner ensemble les bases du débogueur JavaScript conçu pour l'univers Mozilla (navigateurs Mozilla et Firefox, et même le logiciel de courrier électronique Thunderbird !) : **Venkman**. Dans leur tradition de clins d'œil au film *Ghostbusters*, l'équipe Mozilla a en effet nommé son débogueur d'après le personnage de Bill Murray.

Ce débogueur se télécharge à part, sous forme d'une extension. La page officielle est http://www.mozilla.org/projects/venkman/. Le téléchargement est par ailleurs disponible sur le site officiel des extensions et thèmes pour les produits Mozilla, à l'adresse https://addons.mozilla.org/firefox/216/. Vous pouvez également tenter l'installation de sa version francisée : http://extensions.geckozone.org/Venkman. Mais à l'heure où j'écris ces mots celle-ci n'a pas revu sa définition de compatibilité à la hausse, et refusera de s'installer sur un Firefox 1.5.0.1 ou ultérieur (et je ne saurai trop vous conseiller d'être à jour, car il s'agit de mises à jour de sécurité).

Il est certes possible d'ajuster manuellement l'information dans le paquet, mais c'est une manipulation technique un peu délicate, qui n'entre pas vraiment dans le cadre de ce chapitre. J'utiliserai donc pour ce qui suit la version anglaise.

Une fois l'extension installée, elle est accessible par le menu Outils>JavaScript Debugger. Le débogueur est une application un peu lourde, et mettra quelques secondes à s'afficher. Attention ! Sous Firefox 1.5, il semble qu'une fois fermé, il ne puisse plus être ouvert de nouveau à moins de redémarrer Firefox. Peut-être ce bogue aura-t-il disparu à l'heure où vous lirez ces lignes.

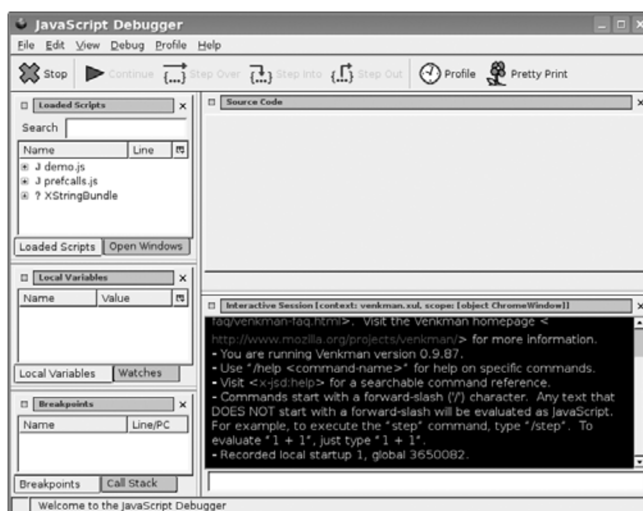


Figure 2–8
Venkman, le débogueur JavaScript

L'interface est composée comme suit :

- En haut, le menu et la barre d'outils. À noter :
- Le menu File permet notamment d'ouvrir une autre page ou un autre script, de sauvegarder et de restaurer l'état des points d'arrêt (*breakpoints*) et des vues espion (*watches*).
- Le menu Debug contient les mêmes commandes que la barre d'outils, mais aussi des sous-menus Error Trigger et Throw Trigger, qui permettent de configurer le débogueur pour qu'il agisse en cas d'erreur ou de lancement d'exception.
- Dans la barre de gauche, pas moins de six vues distinctes, groupées en trois séries d'onglets (mais vous pouvez ajuster comme bon vous semble). On peut afficher ou masquer les vues à loisir, par exemple à l'aide du menu Views>Show/Hide.
- La vue Loaded Scripts affiche tous les scripts chargés ; on est souvent désarçonné par l'apparition des fichiers de script de Firefox lui-même (par exemple `prefcalls.js`). Heureusement, les scripts de votre page apparaissent en haut (ici `demo.js`).
- La vue Open Windows affiche les fenêtres Firefox ouvertes, ce qui vous permet de basculer de l'une à l'autre pour choisir les pages et scripts à déboguer.
- La vue Local Variables affiche les variables locales alors que vous êtes en train d'exécuter un pas à pas dans une fonction. Cela vous évite d'avoir à définir des vues espion pour ces variables.
- La vue Watches affiche les vues espion : des expressions quelconques que vous avez saisies et qui sont évaluées à chaque pas de votre débogage interactif.
- La vue Breakpoints liste les points d'arrêts.
- La vue Call Stack affiche la pile d'appels, c'est-à-dire la séquence des appels imbriqués de fonction qui ont amené à la ligne en cours d'exécution.
- Sur la droite, deux vues se partagent l'espace.
- En haut, la vue Source Code, qui affiche le code source du script sélectionné dans la vue Loaded Scripts, et dans laquelle se passe le pas à pas. C'est aussi là qu'on manipule les points d'arrêt.
- En bas, se trouve la vue Interactive Session, console JavaScript améliorée, dotée d'un interpréteur de commandes spécial (commandes démarrant par `/`). Une liste détaillée des commandes est disponible en cliquant sur le lien qui s'affiche par défaut dans cette console, ou en naviguant directement sur l'URI `x-jsd:help`.

Voici un exemple basique d'utilisation de Venkman, qui est très puissant, mais un peu fouillis, il faut bien le reconnaître. Reprenez l'exemple de code pour les fonctions listant dynamiquement leurs arguments. Je vous conseille de récupérer les fichiers disponibles dans l'archive des codes source pour cet ouvrage, disponible en ligne sur le site des éditions Eyrolles.

Ouvrez le fichier `index.html`, puis ouvrez le débogueur JavaScript. Vous devriez obtenir une vue équivalente à celle de la figure 2-8. Fermez la vue Open Windows, qui ne vous sert à rien, faites glisser vers le bas la séparation entre Source Code et Interactive Session, et videz la session (clic droit dessus ou menu View, option Clear Interactive Session). Double-cliquez à présent sur le script `demo.js` dans la vue Loaded Scripts. Vous devriez obtenir le résultat suivant :

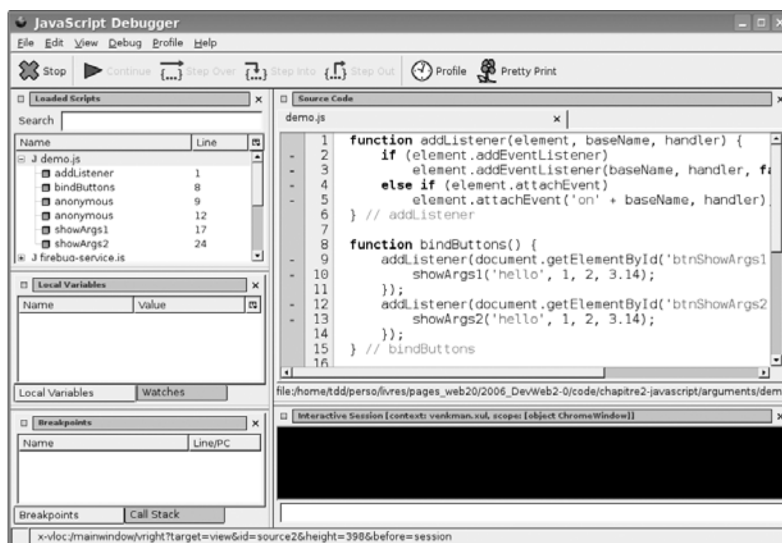


Figure 2–9
Notre script chargé dans un Venkman plus utilisable

Posez à présent un point d'arrêt dans la fonction `showArgs1` : double-cliquez sur le nom de la fonction dans la vue Loaded Scripts, et cliquez dans la marge du code source sur le tiret en face de la ligne 20, ligne soumise à la boucle : vous obtenez un B blanc sur fond rouge, qui indique un point d'arrêt.

Figure 2–10
Le point d'arrêt sur la ligne 20

```
   17  function showArgs1() {
-  18      var msg = '';
-  19      for (var index = 0; index <
B  20          msg += arguments[index]
-  21      alert(msg);
   22  } // showArgs1
```

Reprenez à présent la page web et cliquez sur le bouton Première version. La fonction `showArgs1` est appelée, le point d'arrêt est déclenché, on bascule dans Venkman, et le débogueur passe en mode pas à pas.

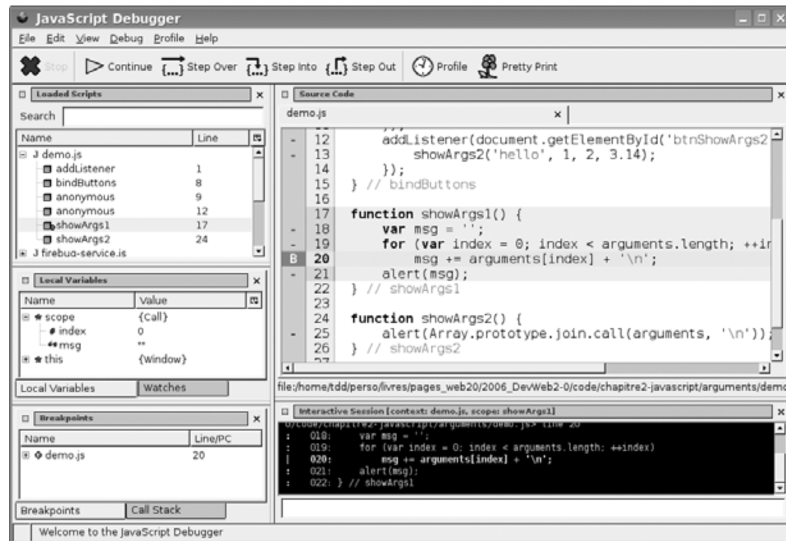


Figure 2–11
Notre débogueur en mode pas à pas

Remarquez plusieurs points :

- La barre d'outils est désormais complètement active, puisque les commandes de pas à pas sont désormais pertinentes.
- La vue `Local Variables` liste la portée courante (`scope`), avec vos variables `index` et `msg`, ainsi que le sens actuel de `this` (on voit qu'il s'agit là de l'objet `window`, et non du bouton sur lequel on a cliqué).
- La ligne courante est mise en exergue (en jaune sur votre écran).
- La vue `Interactive Session` a également signalé l'arrêt, et a affiché un contexte de deux lignes autour de la ligne concernée. Cette vue est un véritable débogueur en ligne de commande, pour les habitués d'outils comme `gdb` notamment.

Nous allons faire un pas à pas de survol (n'entrant pas dans le corps de nos fonctions appelées), ce qu'on appelle en anglais un *step over*. Cliquez sur le bouton `Step over` ou appuyez sur `F12` : vous passez à la ligne suivante, qui est la reprise de la boucle. Une série d'appuis vous mène à travers les tours de la boucle, et vous pouvez suivre dans la vue `Local Variables` l'évolution des variables `index` et `msg`. Vous arrivez finalement au `alert`, qui s'exécute. Repassez ensuite au niveau supérieur, dans la fonction anonyme attachée à l'événement `click` du bouton. Pour continuer l'exécution normalement, plutôt qu'en pas à pas, utilisez la commande `Continue` ou la touche `F5`.

Il s'agit ici de l'utilisation la plus courante du débogueur.

Quelques petites précisions pour finir :

- Les points d'arrêt peuvent être configurés très finement, en précisant par exemple une condition pour leur déclenchement, ou un nombre de passages (*trigger count*) avant qu'ils ne se déclenchent. On peut bien entendu combiner les deux. Pour cela, faites un clique droit sur un point d'arrêt (par exemple sur le B dans la marge) et choisissez en bas du menu surgissant l'option Breakpoint properties :

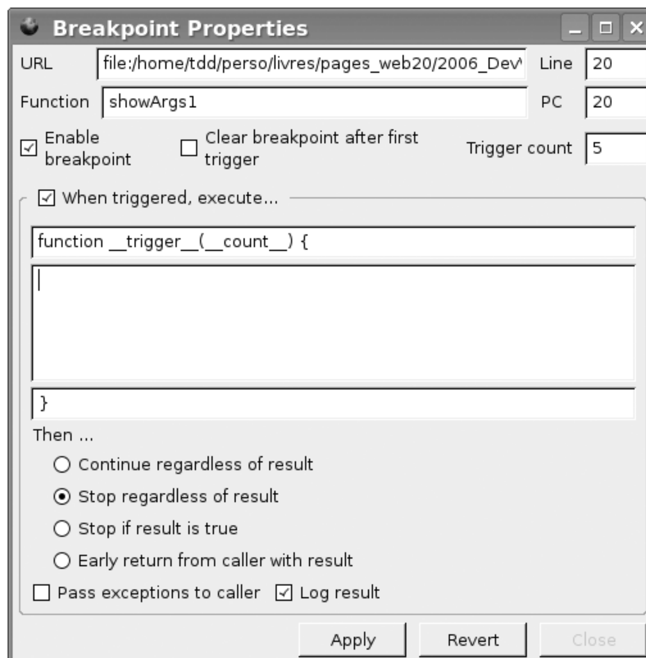


Figure 2–12
Propriétés d'un point d'arrêt : que de possibilités...

- Lorsqu'on est temporairement gêné par un point d'arrêt, plutôt que de le supprimer, il est préférable de le désactiver. Il suffit pour cela de cliquer à nouveau sur le B : il passe alors en F (*Future breakpoint*). Pour le réactiver, cliquez sur le bouton droit et choisissez Set breakpoint.

Venkman dispose d'une documentation exhaustive : n'hésitez pas à la consulter pour en découvrir tous les secrets si vous avez des besoins plus complexes à réaliser.

Firebug, le couteau suisse du développeur Web 2.0

Venkman est souvent jugé trop lourd, ou trop complexe, pour un usage classique. Il se trouve que depuis quelques temps, on dispose d'une alternative, au travers d'une des nombreuses fonctions de l'extension Firebug.

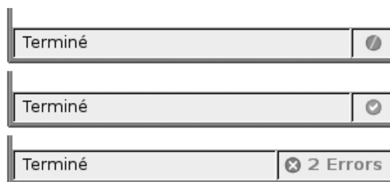
Pour les développeurs web, Firefox propose une pléthore d'extensions sympathiques. Les deux plus populaires, et de loin, sont la Web Developer Toolbar de Chris Pederick, et Firebug de Joe Hewitt.

La première nous permet de triturer la page elle-même, en affichant quantité d'informations utiles et en activant ou désactivant facilement des scripts, feuilles de styles, etc. Elle est immensément utile, mais ne concerne pas spécialement JavaScript ou le DOM.

Firebug fournit une série d'outils dédiés au débogage, comme son nom l'indique. Nous aurons l'occasion de l'utiliser au chapitre 3 pour examiner le DOM de la page (inspecteur DOM alternatif), ainsi qu'aux chapitres 5 et 6 pour examiner le détail de nos requêtes Ajax (une fonctionnalité des plus précieuses !). Pour l'instant, nous allons examiner ses capacités de débogueur JavaScript.

Firebug est installable depuis http://addons.mozilla.org/firefox/1843/. Une fois l'extension installée et Firefox en mode redémarrage, vous disposez en bas de vos pages d'un petit indicateur. Il est gris quand vous n'êtes sur aucune page (par exemple, sur la page vierge d'accueil que vous avez peut-être configurée), vert quand vous êtes sur une page sans erreur JavaScript, et rouge dans le cas contraire.

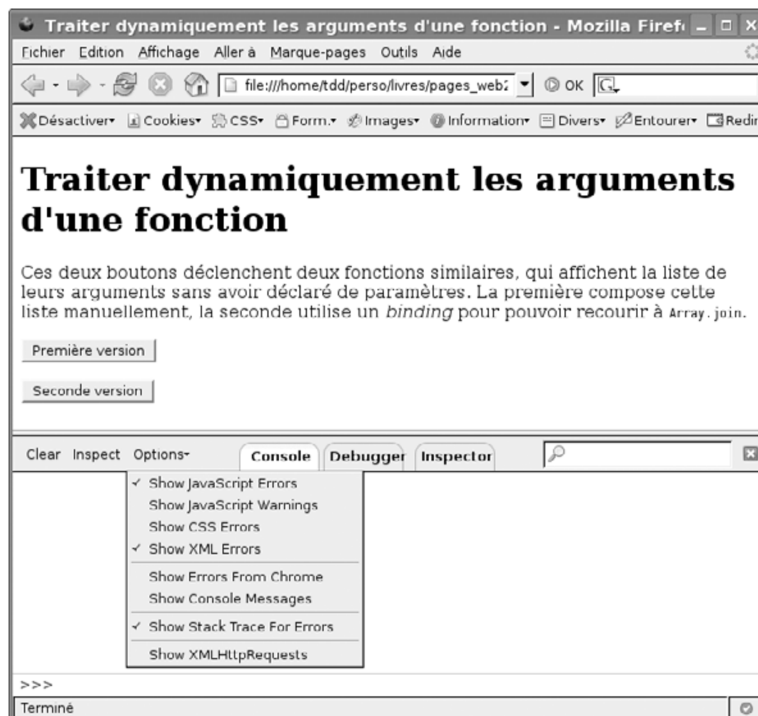
Figure 2–13
Les états de l'indicateur Firebug



En cliquant sur cet indicateur, vous ouvrez la barre Firebug, placée en bas de votre page. On peut aussi l'ouvrir et se placer directement sur la ligne de commande de la console JavaScript en pressant Ctrl+Maj+L. Firebug renferme en effet une console JavaScript fort pratique, interactive, plus utile (et plus jolie !) que celle fournie en standard dans Firefox. Le menu Options de la console Firebug permet de préciser ce qu'on souhaite y voir apparaître (voir figure 2-14). Par défaut, il affiche les erreurs JavaScript (mais pas les avertissements) ainsi que les erreurs XML.

L'ongletDebugger propose un débogueur tout simple, mais largement suffisant dans la plupart des cas. Attention ! Deux débogueurs ne peuvent pas cohabiter dans Firefox. Si vous souhaitez utiliser celui de Firebug, désinstallez d'abord Venkman et relancez Firefox.

Figure 2–14
La console Firebug
et son menu Options



Chargez donc à nouveau notre exemple de fonction listant ses arguments, et cliquez sur l'onglet Debugger. Vous y trouverez le code source de votre script. Firebug ne liste que les scripts chargés par la page, ce qui est plus pratique que le comportement de Venkman. Une liste déroulante Scripts, en bas du débogueur, permet de choisir le script voulu, ce qui est inutile ici. Faites défiler le script pour retrouver la ligne 20, et cliquez dans la marge à gauche du numéro de ligne : vous venez de définir un point d'arrêt. Cliquez à présent sur le bouton Première version de notre page, et le point d'arrêt se déclenche (voir figure 2–15).

Remarquez qu'une vue supplémentaire est immédiatement apparue sur la droite : les variables locales. Par ailleurs, juste en dessous, vous trouverez la pile d'appels sous forme d'une liste déroulante. Pour progresser en pas à pas de survol (*step over*), utilisez le premier bouton à flèche coudée ou Ctrl+Alt+Maj+Droite (une véritable combinaison de touches !).

Les autres raccourcis sont décrits dans le menu Outils>Firebug, et sont logiques malgré leur complexité : ils débutent tous par Ctrl+Alt+Maj, après quoi Droite fait un *step over*, Bas un *step into* (entre dans nos fonctions lorsqu'on les appelle) et Haut un *step out* (termine la fonction courante et en ressort).

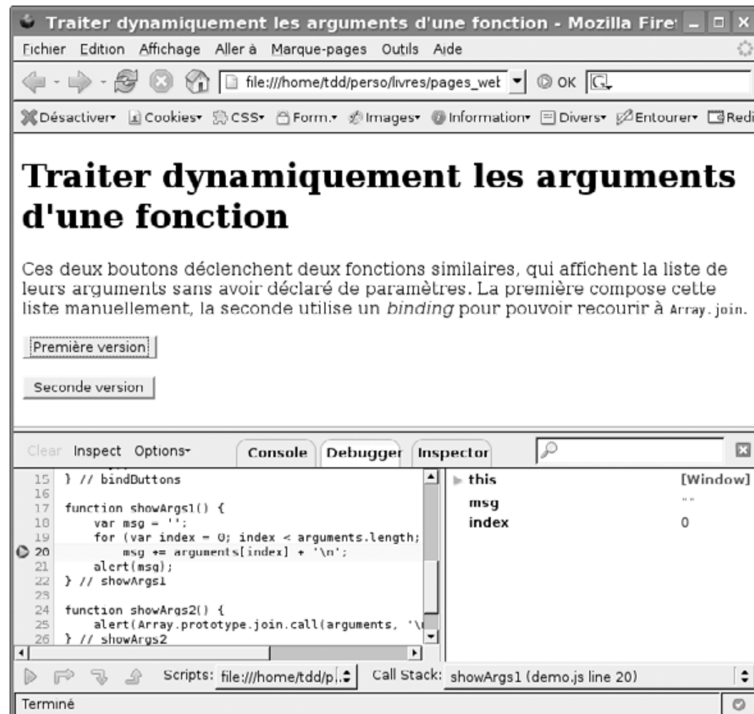


Figure 2–15
Déclenchement du point d'arrêt dans le débogueur de Firebug

Pour simplement continuer l'exécution jusqu'à la fin (ou jusqu'au prochain point d'arrêt), cliquez sur la flèche bleue ou faites Ctrl+Alt+Maj+Gauche.

Firebug regorge de possibilités, et en terme de JavaScript, il ne s'arrête pas là. Si vous souhaitez simplement permettre à vos scripts de vous signaler quelque chose pendant le développement, Firebug met à votre disposition un objet global nommé `console`, avec cinq méthodes : `error`, `warn`, `info`, `debug` et `log`. Elles envoient le message passé en paramètre, avec le bon type, dans la console de Firebug (voir figure 2–16).

Par ailleurs, Firebug met à disposition toute une série de méthodes pour vos tests unitaires, appelées assertions (le terme est familier à quiconque a déjà fait des tests unitaires, dans quelque langage que ce soit). On trouve aussi des méthodes pour aider à la mesure du temps d'exécution de fragments de code, à l'affichage de textes à paramètres formatés, ou à l'examen d'objets.

Enfin, sachez que des fonctions courtes, dans le style de Prototype, sont disponibles pour aller rapidement chercher un élément par son ID, faire une extraction CSS ou encore une extraction XPath, directement depuis la console !

Vous en saurez davantage sur la page de documentation de ces possibilités dans Firebug : http://joehewitt.com/software/firebug/docs.php.

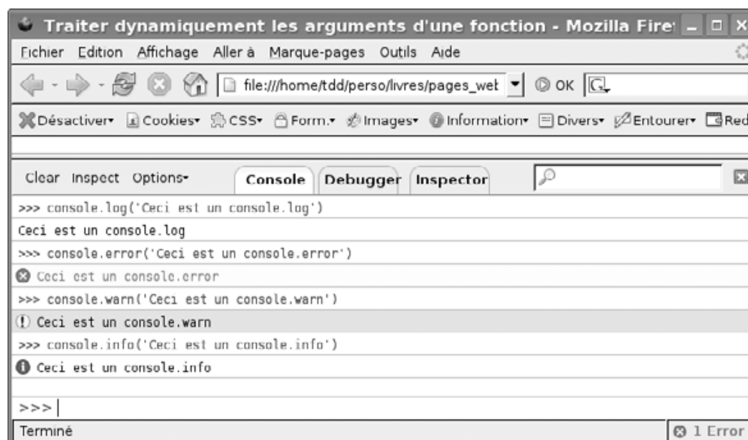


Figure 2–16
Exemples d'utilisation de l'objet console avec Firebug

Pour aller plus loin

Livres

JavaScript : The Definitive Guide (5e édition)
David Flanagan
O'Reilly Media, août 2006, 1 018 pages
ISBN 0-596-10199-6

La référence historique, avec tous les détails du langage, et une pléthore d'exemples. Existe aussi en français dans sa 4e édition :

JavaScript (4e édition)
David Flanagan
O'Reilly, septembre 2002, 955 pages
ISBN 2-841-77212-8

JavaScript Bible (5e édition)
Brendan Eich
Wiley Publishing, mars 2004, 1 236 pages
ISBN 0-764-55743-2

L'autre référence, qui possède l'avantage d'être écrite par l'inventeur du langage, lequel continue de piloter son évolution. En revanche, ne dispose a priori pas d'une version française...

PPK on JavaScript
Peter-Paul Koch
New Riders Publishing, août 2006, 400 pages
ISBN 0-321-42330-5

Un ouvrage tout récent et bien fait par un des principaux gourous du langage. Décrit notamment en détail les interactions entre JavaScript et CSS.

Les expressions régulières par l'exemple
Vincent Fourmond
Eyrolles, août 2005, 126 pages
ISBN 2-914010-65-6

Une bonne façon de se faire les dents sur ce sujet qui fait injustement peur, et dont les bénéfices sont pourtant considérables !

Sites

- La *Mozilla Developer Connection* regorge de ressources précieuses sur JavaScript et le DOM. Je vous conseille en particulier leur guide et leur référence de JavaScript 1.5. Une partie a été traduite en français.
  - http://developer.mozilla.org/en/docs/JavaScript
  - http://developer.mozilla.org/fr/docs/JavaScript
- Rien ne vaut la spécification officielle pour vérifier un point de détail, ou déterminer si MSIE est fautif ou non, mais attention, c'est... aride :
  http://www.ecma-international.org/publications/standards/Ecma-262.htm
- Brendan Eich est l'inventeur et le chef de projet actuel du langage. Son blog permet de suivre l'actualité de JavaScript et de goûter aux futures améliorations :
  http://weblogs.mozillazine.org/roadmap/
- Peter-Paul Koch, dont le livre figure plus haut :
  http://www.quirksmode.org
- Venkman, le débogueur JavaScript aux enzymes, est disponible comme extension :
  https://addons.mozilla.org/firefox/216/
- Firebug, le couteau suisse du développeur Web 2.0, est lui aussi une extension :
  https://addons.mozilla.org/firefox/1843/

3

Manipuler dynamiquement la page avec le DOM

JavaScript est au développeur web ce que les outils sont à l'artisan : le moyen de travailler. Encore faut-il avoir quelque chose sur quoi travailler. Pour un développeur web qui cherche à rendre sa page vivante, cette matière, c'est le DOM, le *Document Object Model*, un ensemble d'interfaces permettant aux langages de programmation d'accéder aux objets qui composent le document. C'est via ces interfaces qu'on peut manipuler efficacement la page.

Dans ce chapitre, nous allons voir qu'il n'y a pas un DOM mais plusieurs, classés par niveau et par thème. La clé du succès résidant dans la connaissance des quelques interfaces fondamentales, nous consacrerons quelques pages à découvrir celles-ci, en nous concentrant sur les aspects noyau et HTML. Fidèle à l'esprit de l'ouvrage, nous soulignerons quelques pratiques recommandées, avant d'explorer dans le détail la gestion des événements du document. Pour finir, nous construirons quelques exemples concrets très souvent utiles.

Pourquoi faut-il maîtriser le DOM ?

Il est aisé de croire qu'on peut se passer d'étudier les détails du DOM, en particulier lorsqu'on a l'intention de recourir à des bibliothèques ou frameworks tels que Prototype, script.aculo.us ou encore Dojo. C'est une attitude dangereuse, qui peut entraîner bien des heures perdues au débogage, simplement parce qu'on aura voulu faire l'économie de quelques heures en amont.

La pierre angulaire des pages vivantes

On l'a vu en introduction, il n'y a pas de pages vivantes sans le DOM. Ceux qui en sont encore à manipuler leurs pages « à l'ancienne », ce qu'on appelle aujourd'hui le DOM niveau 0, sont condamnés au recyclage : ces possibilités ne font l'objet d'aucun standard et sont déjà, pour ainsi dire, en préretraite. Gardez-vous de subir le même sort en vous accrochant à des techniques obsolètes !

Ce serait d'autant plus dommage qu'en soi, le DOM est assez simple. Malgré une certaine verbosité et une pléthore d'interfaces, la complexité n'est qu'apparente, et nous avons pour une fois la chance d'avoir un standard plutôt cohérent dans l'ensemble, et bien pris en charge par les principaux navigateurs. Une fois certaines règles et certains concepts assimilés, l'utilisation est sans surprise (à l'exception, comme toujours, de certains comportements sur MSIE, que nous aborderons spécifiquement).

Maîtriser la base pour utiliser les frameworks

Quand bien même vous envisagez d'utiliser des bibliothèques facilitant grandement votre travail, comme Prototype, voire des frameworks entiers tels que script.aculo.us ou Dojo, cela ne vous dispensera pas de connaître un minimum le DOM.

De nombreuses fonctions de ces bibliothèques sont documentées en utilisant le vocabulaire du DOM : ID d'élément, nœuds fils, nœud père, collection, nœuds textuels sont autant de concepts DOM qui sont utilisés à tout bout de champ par les bibliothèques de plus haut niveau. Sans maîtriser au moins les bases, vous risquez fort de vous y perdre.

Comprendre les détails pour pouvoir déboguer

Les bibliothèques ne sont d'ailleurs pas exemptes de bogues, loin s'en faut. Même lorsque vous utilisez des fonctionnalités stables, rien ne vous empêche de mal interpréter le sens d'un argument, d'en oublier un autre, ou de mal satisfaire aux exigences d'une fonction.

Lorsque le problème résultant va faire surface, vous n'aurez d'autre choix que de démarrer une séance de débogage ; mais même avec de bons outils à l'appui (débogueur JavaScript et inspecteur DOM sont des incontournables), vous serez bien en peine de comprendre tant le code JavaScript que l'arbre DOM examinés si vous avez négligé d'étudier un peu DOM dans le détail. Pour quelques heures économisées à tort, vous voilà face à de nombreuses heures fastidieuses et peut-être quelques cachets d'aspirine.

En somme, je ne saurais trop vous recommander de consacrer un peu de temps à lire la suite de ce chapitre. Vos pages web vous le rendront au centuple.

Le DOM et ses niveaux 1, 2 et 3

Le DOM est un standard aux multiples facettes. Non seulement il évolue dans le temps, au travers de niveaux (qui tiennent lieu de versions), mais en plus il est modularisé par thème, ou aspect si vous préférez.

Vue d'ensemble des niveaux

Voici un tour d'horizon des niveaux successifs du DOM. Les spécifications W3C indiquées peuvent être arides à consulter pour le débutant : si vous voulez vous y plonger rapidement, vous gagnerez à lire d'abord les conseils de l'annexe C.

Tableau 3–1 Les niveaux du DOM

Niveau	Description
0	Il ne s'agit pas d'un standard à proprement parler, mais du nom sous lequel on désigne aujourd'hui les possibilités de manipulation du document sans prise en compte du DOM, telles qu'on en trouve encore dans de trop nombreux didacticiels et ouvrages (tous datant un peu, il est vrai). Par exemple, le code suivant : `document.forms['inscription'].nom_client.value` ...est très clairement du niveau 0 : la collection `forms` ne propose pas d'opérateur `[]` dans le DOM, même si sa fonction `namedItem` a un sens équivalent (et encore, pas en XHTML). Quant à l'utilisation d'un objet fils nommé d'après l'attribut `name` du champ, elle est totalement absente du DOM également : on préfère accéder directement au champ par son ID, où qu'il soit dans le document. Peter-Paul Koch propose un excellent article sur ce niveau : http://www.quirksmode.org/js/dom0.html.
1	Le niveau est le premier véritable standard du DOM, remontant à 1998 (2000 pour une révision jamais finalisée). Il n'était pas encore découpé en modules et fournissait déjà les principales interfaces pour le noyau et HTML. Dernière révision : http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/.

Tableau 3–1 Les niveaux du DOM (suite)

Niveau	Description
2	L'ensemble des modules du niveau 2 date du 13 novembre 2000, mais le module HTML a subi cinq révisions depuis, la dernière remontant à janvier 2003. Ce niveau ajoute principalement la prise en charge de XHTML 1.0, avec entre autres impacts une déclinaison de nombreuses méthodes pour gérer les espaces de noms. L'interface `HTMLOptionsCollection` fait son apparition. Le niveau 2 est celui le plus largement pris en charge par les principaux navigateurs actuels. Nous nous intéresserons principalement aux modules noyau, événements et HTML : • http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/ • http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/ • http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/
3	Le travail actuel porte sur le niveau 3, état de l'art du DOM. Il modifie principalement les aspects noyau, vues/formatage et événements, et ajoute plusieurs modules : Load and Save, Validation et XPath. En regard des possibilités déjà existantes, c'est surtout ce dernier module qui ajoute de la valeur, à mon sens. Nous l'utiliserons d'ailleurs au chapitre 5, lorsque nos requêtes Ajax récupéreront un contenu XML. Ce niveau vise surtout à ajouter des possibilités de l'univers XML à nos pages (comparaison de nœuds et d'arbres, gestion de l'encodage, des URI relatifs, de la normalisation, de la validité vis-à-vis des schémas et DTD), et à simplifier certaines opérations (par exemple grâce à `adoptNode`, `renameNode`, `textContent` et `wholeText` ou aux mécanismes permettant d'associer des données utilisateur aux nœuds). La gestion des événements est aussi mieux spécifiée. Certains modules sont déjà finalisés : Noyau, Load and Save et Validation. Les autres sont encore en travaux. Ceci dit, les dernières révisions datent d'avril 2004, le W3C faisant là aussi la preuve de son extraordinaire rapidité... On citera principalement le module `noyau` : http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

Support au sein des principaux navigateurs

Comme on a pu le voir à l'avant-propos, les principaux navigateurs prennent plutôt bien en charge le DOM dans son niveau 2. C'est le cas de Mozilla, Firefox, Camino, Safari, Konqueror et Opera, pour ne citer qu'eux. MSIE dispose aussi d'une bonne prise en charge, malgré quelques écarts hélas problématiques, notamment pour la gestion des listes dans les formulaires (on y reviendra dans le détail plus tard).

Certains fournissent partiellement le niveau 3, comme Firefox et donc Camino (notamment le module XPath et les possibilités XSLT !), Opera (XPath et Load and Save) et Konqueror. Côté MSIE, il ne faut pas s'y attendre avant au moins la version 8, soit dans plusieurs années...

Les aspects du DOM : HTML, noyau, événements, styles...

Nous l'avons déjà évoqué, le standard DOM est aujourd'hui divisé en modules, ou aspects, afin de mieux structurer son contenu et de faciliter la consultation. En voici un rapide tour d'horizon (les modules marqués d'une astérisque ne disposent pas d'une version finalisée à l'heure où j'écris ces lignes) :

Tableau 3–2 Modules du DOM avec leur niveau d'apparition et leur dernière révision

Module	Apparu dans le niveau	Dernière version	Description
Noyau (*Core*)	1	07/04/2004	Interfaces fondamentales (voir prochaine section).
HTML	1	09/01/2003	Interfaces spécifiques aux contenus des documents (X)HTML.
Vues (*Views*)	2	13/11/2000	Manipulation d'une représentation visuelle spécifique d'un document (par exemple une fois qu'une feuille CSS aura été appliquée).
Événements (*Events*)	2	13/11/2000	Gestion événementielle (écoute, traitement, propagation...).
Style	2	13/11/2000	Manipulation des feuilles de styles et des styles calculés pour chaque nœud.
Traversal and Range	2	13/11/2000	Parcours et filtrage d'un document (*traversal*) et manipulation de fragments de documents (*range*).
Load and Save	3	07/04/2004	Stockage et récupération de documents par (dé)sérialisation.
Validation	3	15/12/2003	Vérification de la conformité du document à sa grammaire (DTD/schéma).
XPath*	3	26/02/2004	Extraction de nœuds du document à l'aide de la syntaxe XPath.
Views and Formatting*	3	26/02/2004	Extensions au module Vues.
Abstract Schemas*	3	27/07/2002	Manipulation des grammaires de document (DTD, schéma, etc.).

Maîtriser les concepts : document, nœud, élément, texte et collection

Il faut d'abord comprendre que le DOM au sens W3C s'applique à un document de type XML. Cela n'implique pas forcément que la syntaxe du document soit conforme à XML (même si c'est généralement le cas), mais que le document résultat soit d'une nature identique : un arbre de nœuds imbriqués les uns dans les autres.

Par conséquent, dans le cadre de pages web, on pourra exploiter le DOM d'autant plus efficacement (et avec moins de surprises potentielles) qu'on utilise un balisage

conforme à XHTML 1 strict. Un balisage plus laxiste, moins cohérent, du type de HTML 4.01, peut engendrer un DOM parfois... inattendu !

Si ces notions de balisage sémantique et de différences entre XHTML et HTML vous semblent un peu confuses, n'hésitez pas à faire un tour à l'annexe A.

Le DOM de votre document : une arborescence d'objets

Un DOM est donc une arborescence d'objets, ou plus généralement de nœuds, qui représente le document, c'est-à-dire, dans le cas qui nous concerne, la page web. On accède au document lui-même au travers d'une interface `Document`. Lorsqu'on part de là pour explorer le contenu du document, on ne tombe que sur des nœuds, mais chacun d'un type spécifique.

Les principaux types de nœuds reflètent les différentes catégories de balisage possibles dans un document XML (donc XHTML) : éléments, attributs, commentaires, textes (ce qu'on met généralement entre la balise ouvrante et la balise fermante), mais aussi des types moins courants, comme « instruction », le type des déclarations `DOCTYPE` par exemple.

Tous ces types partagent un ensemble de propriétés et fonctions héritées de leur type générique : le nœud (*node*). Mais ils ont aussi leurs spécificités, représentées par leurs interfaces dédiées (par exemple, `Element` pour les éléments, ou `Attr` pour les attributs).

L'arborescence du DOM est souvent bien plus verbeuse que le balisage XHTML correspondant, et c'est souvent une source de surprise pour les débutants. Si on n'y prend pas garde, elle peut causer des confusions responsables de bien des bogues dans les scripts exploitant le DOM.

Voici un document XHTML d'exemple :

Listing 3-1 Un document XHTML simpliste

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Un document tout petit</title>
</head>
<body>
<h1>Un document tout petit</h1>
<p>Ceci est un petit document</p>
</body>
</html>
```

Derrière ce document se cache une arborescence DOM plus épaisse qu'on ne pourrait croire !

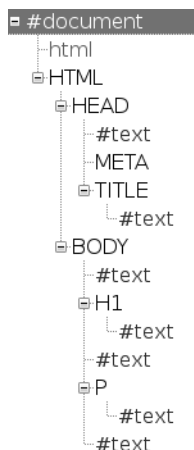


Figure 3–1
Le DOM de ce document, vu par l'inspecteur DOM de Firefox

On voit bien que la racine est un objet `Document` ; le `html` minuscule représente un nœud de type instruction, qui correspond à la déclaration `DOCTYPE` en haut du code source (`html` est le nom de l'élément racine défini par la déclaration). Mais que sont tous ces nœuds `#text` ?! Et pourquoi les balises sont-elles en majuscules ?

Le plus simple d'abord : les balises apparaissent en majuscules parce que, même si XHTML exige des balises minuscules (car XML est sensible à la casse, et la DTD définit les balises en minuscules), le DOM utilise généralement les noms canoniques des balises, ce qui implique généralement qu'ils soient en majuscules. Finalement, la raison d'être des `#text` n'est guère plus compliquée. Comme nous le verrons à nouveau plusieurs fois dans les sections suivantes, consacrées aux principales interfaces du DOM noyau et HTML, une balise non vide (c'est-à-dire une balise disposant de versions ouvrante et fermante, et ayant au moins un caractère entre les deux) ne contient pas directement le texte qui y figure. On a ici deux nœuds : un nœud de type élément pour la balise à proprement parler (parties ouvrante et fermante), et un nœud de type texte pour le texte figurant entre les deux.

Prenons le fragment suivant :

```
<title>Un document tout petit</title>
```

Dans le DOM, ce fragment est représenté par deux nœuds, comme on peut le voir sur la figure 3-1 :

1 Un nœud élément nommé `TITLE`, sans valeur.
2 Un nœud texte sans nom (d'où le `#text`), de valeur `Un document tout petit`.

Ce que nous prenons pour des espaces sans signification particulière – sous prétexte qu'un navigateur va représenter la plupart des séries d'espaces (et retours chariot, tabulations, etc.) sous forme d'un seul espace à l'affichage – ne disparaît pas pour autant du document, et se retrouve donc dans le DOM. Ainsi, les nœuds texte figurant avant `H1`, entre `H1` et `P`, et après `P`, représentent les lignes vides entre ces éléments.

Nous verrons un peu plus tard, en parlant de l'interface `Text`, que le type MIME du document (principalement `text/html` ou `application/xhtml+xml`) influe sur les caractères qu'on trouve dans les nœuds texte ainsi que sur les ajustements que le navigateur peut réaliser à la création du DOM. Par exemple, si vous regardez attentivement la figure 3-1 et la comparez au listing 3-1, vous verrez qu'il manque a priori un nœud texte entre `META` et `TITLE`.

Voyons à présent une présentation des principales interfaces, en commençant par celles du module noyau.

Node

C'est l'une des deux interfaces incontournables du DOM. Tous les nœuds d'un document, quel que soit leur type spécifique, sont avant tout des nœuds. Chaque interface dédiée hérite donc de l'interface `Node`. Du coup, toutes les interfaces sur nœuds que vous manipulerez (éléments, attributs, textes...) disposeront des propriétés et méthodes de `Node`.

On peut donc tout faire rien qu'en utilisant ces méthodes, mais lorsqu'on manipule des éléments, on utilisera généralement les méthodes de plus haut niveau de l'interface `Element`, notamment pour manipuler les attributs (sinon, c'est l'enfer !). Et dans le cadre précis de pages web, on aura recours à toutes les propriétés prédéfinies pour chaque type de balise.

Voici les quatre propriétés fondamentales de `Node`.

Tableau 3–3 Propriétés fondamentales de Node

Propriété	Description
`nodeName`	Le nom du nœud. Il varie suivant le type du nœud. Cas principaux : pour un élément, équivalent de la propriété `tagName` ; pour un nœud texte, `#text` ; pour un attribut, équivalent de la propriété `name`.

Tableau 3–3 Propriétés fondamentales de Node (suite)

Propriété	Description
`nodeType`	Le type du nœud. Valeur numérique indiquant le type du nœud (élément, texte, attribut, etc.). Les navigateurs offrant une prise en charge propre fournissent les constantes pour l'ensemble des types, tandis que les autres vous laissent vous débrouiller avec les valeurs numériques. Voir plus bas pour les principales valeurs.
`nodeValue`	Valeur du nœud. Elle varie suivant le nœud. Pour un attribut, équivalent de la propriété `value` ; pour un texte, un commentaire ou une section CDATA (eh bien… le texte !) ; pour la plupart des autres types, `null`.
`attributes`	On se demande ce que cette propriété fait là plutôt que dans l'interface `Element`, puisqu'elle n'est définie que pour les nœuds de type élément ! Il s'agit d'un tableau à valeurs nommées (interface `NamedNodeMap`, vue plus bas) fournissant tous les attributs de l'élément. On ne s'en sert que pour lister les attributs dynamiquement : pour un attribut précis, on utilisera `getAttribute` et `setAttribute`, définis dans l'interface `Element`, et bien plus pratiques.

Les principales constantes pour les types de nœud sont (valeur et nom) :

1 `Node.ELEMENT_NODE`

2 `Node.ATTRIBUTE_NODE`

3 `Node.TEXT_NODE`

Voyons à présent les propriétés qui permettent de se déplacer dans le DOM, tâche absolument critique pour la plupart des scripts.

Tableau 3–4 Propriétés de parcours du DOM de Node

Propriété	Description
`parentNode`	Le nœud père dans l'arborescence. Il aura pour valeur `null` si le nœud vient d'être créé ou s'il s'agit du nœud `Document`, tout en haut de l'arborescence, d'un attribut ou de quelques autres types ésotériques. Dans le cas contraire, indique le nœud parent, ou conteneur, du nœud courant. Par exemple, le nœud père de `title` est normalement `head`, celui de `head` est `html`, celui de `html` est `#document` et la chaîne s'arrête là.
`childNodes`, `firstChild`, `lastChild`	En descente maintenant : ces propriétés permettent d'accéder aux nœuds fils du nœud courant. Un élément vide (`<balise … />`) n'aura pas de nœuds fils, pas plus qu'un attribut, un commentaire ou une instruction. Dans un tel cas, `childNodes` renverra une liste vide (`0 == childNodes.length`), et `firstChild` comme `lastChild` renverront `null`. Sinon, la liste n'est pas vide, `firstChild` renvoie le premier nœud fils et `lastChild` le dernier nœud fils. S'il n'y a qu'un nœud fils, on a donc `firstChild == lastChild` !
`previousSibling`, `nextSibling`	Renvoient les nœuds frères précédent et suivant, respectivement. Un nœud frère est un nœud ayant le même nœud parent que le nœud courant, donc figurant au même niveau dans le document. Les notions de précédent et suivant respectent l'ordre du document.

Pour bien fixer les idées, voici un petit fragment de XHTML et quelques résultats d'expressions. On utilisera dans le code source JavaScript des variables nommées d'après les attributs `id` du source XHTML (ce qui ne se fait pas tout seul : on supposera qu'on a correctement défini ces variables auparavant).

```
<h1 id="header">Personnes inscrites</h1>
<ul id="people">
    <li id="al">Alexis</li>
    <li id="nioute">Anne-Julie</li>
    <li id="elodie">Élodie</li>
    <li id="mimi">Marie-Hélène</li>
    <li id="xavier">Xavier</li>
</ul>
```

On a alors :

```
header.nextSibling.nodeType == Node.TEXT_NODE // Surprise !
header.nextSibling.nextSibling == people
nioute.lastChild == nioute.firstChild
elodie.nextSibling == mimi
people.previousSibling == header
header.childNodes.length == 1
header.firstChild.nodeType == Node.TEXT_NODE
// Ci-dessous : 6 noeuds texte entrelacés à 5 éléments
people.childNodes.length == 11
people.firstChild.nodeType == '#text'// Surprise !
people.firstChild.nodeValue == '    '// En mode HTML en tout cas
elodie.firstChild.nodeName == '#text'
xavier.lastChild.nodeValue == 'Xavier'
xavier.parentNode.firstChild.nextSibling == al
// Ci-dessous, valide en mode HTML. En mode XML, serait 'h1'
nioute.parentNode.previousSibling.nodeName == 'H1'
```

On a souvent tendance à oublier ces maudits nœuds texte un peu partout, dus à l'indentation, aux retours à la ligne... Avec un peu d'habitude, on finit par s'y habituer, et arrêter de voir surgir des erreurs JavaScript du type « pas de propriété xxx pour le nœud ». Et surtout, avec les méthodes étendues de l'objet `Element` dans Prototype (que nous étudierons au chapitre 4), on se déplace beaucoup plus simplement et efficacement !

Certains écrivent rapidement des fonctions du style `firstElementChild`, `lastElementChild`, `previousElementSibling` et `nextElementSibling`, pour éviter d'avoir à penser à cela, mais il s'agit à mon sens d'une fausse bonne idée. En effet, le simple fait d'utiliser nos fonctions propres plutôt que les fonctions natives du DOM nous amène à penser au problème ! Qui plus est, cela crée une dépendance plutôt superflue à nos quatre petites fonctions, que nous devrons d'ailleurs utiliser comme

des fonctions classiques (`x = firstElementChild(node)`) plutôt que comme des propriétés (`x = node.firstChild`). Le jeu n'en vaut pas la chandelle. Pour les besoins complexes de parcours, on utilisera simplement des mécanismes plus avancés que ces propriétés.

Après avoir passé en revue les propriétés, examinons les méthodes de `Node`, qui sont très fréquemment utilisées, puisqu'elles constituent le seul moyen d'altérer le DOM en ajoutant, déplaçant ou retirant des nœuds.

Tableau 3–5 Principales méthodes de Node

Méthode	Description
`appendChild(newChild)`	Ajoute le nœud `newChild` en dernière position à l'intérieur du nœud courant (`newChild` devient automatiquement le `lastChild`). Astuce : si `newChild` était déjà présent dans le DOM, il est automatiquement retiré d'abord. Un déplacement se fait donc en un seul appel de méthode.
`cloneNode(deep)`	Crée un clone du nœud courant, sans l'attacher au DOM. Le paramètre `deep` est un booléen indiquant s'il s'agit d'une copie profonde (incluant tous les nœuds fils, donc une copie de l'arbre dont le nœud courant est racine) ou d'une copie superficielle (juste le nœud courant, sans ses nœuds fils). Ce dernier cas est très rare, on fait donc généralement un `cloneNode(true)`. En somme, c'est un peu la même chose qu'un constructeur copie.
`hasAttributes()`	Indique si le nœud (qui doit être un élément) a des attributs. Méthode introduite au niveau 2, et cohérente avec la présence de la collection `attributes`. Équivalent propre de `0 != attributes.length`.
`hasChildNodes()`	Permet de savoir si le nœud a des nœuds fils. Équivalent propre de `0 != childNodes.length`.
`insertBefore(new, ref)`	Insère un nœud fils à une position bien déterminée. Le nœud à insérer est passé en premier (`new`), le nœud de référence en second. Comme le nom de la méthode l'indique, `new` sera positionné avant `ref`. Ainsi, pour insérer `n` comme premier fils de `p` : `p.insertBefore(n, p.firstChild)`. Notez que l'insertion peut avoir lieu même quand `p` n'a pas encore de nœud fils, car `p.insertBefore(n, null)` est équivalent à `p.appendChild(n)`. Comme pour `appendChild`, si `new` était déjà présent dans le DOM, il est automatiquement retiré d'abord.
`removeChild(old)`	Retire le nœud fils `old` du DOM. C'est le seul moyen de retirer un nœud : on n'a pas de méthode `remove` ou `delete` qui supprime le nœud sur lequel elle est invoquée : il faut demander à son nœud parent (attention à ne pas passer `null`). Notez bien que je dis retirer, pas détruire. Le nœud existe toujours en mémoire, vous pouvez garder une référence dessus. Je rappelle que JavaScript n'a pas de destructeurs d'objets, vu qu'il s'agit d'un langage « à ramasse-miettes » (*garbage collector*).
`replaceChild(new, old)`	Remplace le nœud fils `old` par le nœud `new`. Renvoie `old`. Comme pour `appendChild` et `insertBefore`, si `new` était déjà présent dans le DOM, il est automatiquement retiré d'abord.

Petite précision, évidente à l'utilisation, mais sait-on jamais : tous les ajouts, retraits et remplacements concernent le nœud (`new` ou `old`) et ses nœuds fils, c'est-à-dire le nœud en tant que fragment de document.

Avant de pouvoir manipuler ces méthodes dans des exemples, nous allons devoir explorer l'interface `Document`, qui nous permettra de mettre la main sur des éléments existants (pour les déplacer, les modifier ou les retirer du DOM), et d'en créer de nouveaux pour les ajouter au DOM.

Notons enfin que le DOM niveau 3 a rajouté quelques méthodes et propriétés de confort à l'interface `Node`, en particulier `isSameNode`, `isEqualNode`, `getUserData`, `setUserData` et `textContent`. Il s'agit ici surtout de raccourcir le code nécessaire à quelques tâches courantes.

Document

C'est l'autre interface incontournable, quoi que vous vouliez faire avec le DOM. C'est grâce à elle qu'on accède au document, comme son nom l'indique. Elle est la plupart du temps fournie par un objet JavaScript appelé `document`. Cet objet a d'ailleurs la bonne idée d'implémenter aussi l'interface `DOMImplementation`, que nous verrons plus loin. En somme, tout fragment de script DOM utilise `document`.

Voici les principales propriétés et méthodes.

Tableau 3–6 Principales propriétés et méthodes de Document

Propriété/méthode	Description
`documentElement`	L'élément (interface `Element`) racine du document. Dans un document (X)HTML, c'est l'élément `html`. Pratique pour réaliser un parcours manuel en partant de la racine.
`createElement(tagName)`	Le seul et unique moyen de créer un élément en JavaScript, donc d'enrichir ensuite le document. On passe le nom de la balise en argument. Je conseille les minuscules, qui fonctionneront aussi sur un document 100 % XML, alors que la forme canonique (majuscule) échouerait. Renvoie le nœud fraîchement créé (interface `Element`). Attention : de nombreux exemples en ligne passent un fragment HTML complet, du style `document.createElement('<div id="test">Bonjour</div>');`. C'est une extension partielle de MSIE, ce qui ne fait absolument pas partie du standard.
`createTextNode(data)`	Le seul moyen de créer un nœud texte, l'autre grand type de nœud dans une page web. On passe le texte du nœud en argument. Nous renvoie le nœud texte (interface `Text`) ainsi construit.

Tableau 3–6 Principales propriétés et méthodes de Document (suite)

Propriété/méthode	Description
`getElementsByTagName(tag)`	Très pratique, permet de récupérer une liste (interface `NodeList`) de tous les éléments, dans l'ordre du document (parcours en profondeur, de haut en bas), ayant passé le nom en argument. Là aussi, on est sensible à la casse lorsqu'on est en mode XML. Par exemple, pour récupérer tous les éléments `table` d'une page, on utilisera `document.getElementsByTagName('table')`. La valeur spéciale * est parfois utile et permet de récupérer tous les éléments du document.
`getElementById(id)`	Recherche dans tout le document le nœud dont l'ID (attribut `id`) est fourni. Renvoie l'élément (interface `Element`) en cas de succès, `null` sinon. C'est la méthode incontournable. Inutile d'espérer faire un script utile sans elle. Pour beaucoup, ne pas en disposer revient à ne pas disposer du DOM, et par extension, tester qu'on a le DOM revient à tester, en fait, qu'on a au moins le niveau 2.

Deux mots sur les méthodes `create`*`Xxx`*. D'abord, les nœuds sont bien créés en mémoire et nous sont renvoyés pour manipulation ultérieure, mais ils ne figurent pas encore dans le DOM, ils ne sont rattachés à aucune portion existante du document. Il nous appartient de les insérer à l'endroit qui nous intéresse, avec les méthodes de l'interface `Node`. Ensuite, pour `createElement`, si le type d'élément indiqué prévoit des valeurs par défaut à ses attributs, les nœuds `Attr` correspondants sont automatiquement créés et ajoutés à sa liste `attributes`.

On trouve également les variantes à suffixe `NS` de nombreuses méthodes, qui prennent en charge un paramètre supplémentaire indiquant un espace de noms. On est alors clairement en XHTML, car il s'agit d'une fonction importante héritée du XML, qui permet de faire cohabiter plusieurs vocabulaires dans un même document (par exemple, pour utiliser des balises MathML ou SVG).

Le niveau 3 a ajouté quelques méthodes bien utiles, comme `adoptNode`, `normalizeDocument` et `renameNode`, qui facilitent des tâches assez fréquentes mais auparavant un brin verbeuses à programmer.

Voici à présent quelques exemples, pour se faire une première idée.

Listing 3-2 Un script plutôt indécis qui illustre beaucoup de choses

```
// Création d'un équivalent <h1>Bonjour</h1>
var header = document.createElement('h1');
header.appendChild(document.createTextNode('Bonjour'));

// Ajout au début de <body>
var body = document.getElementsByTagName('body').item(0);
body.insertBefore(header, body.firstChild);
```

```
// Finalement plutôt après l'en-tête d'ID 'main'...
var mainHdr = document.getElementById('main');
mainHdr.parentNode.insertBefore(header,
    mainHdr.nextSibling);

// Et puis carrément, en remplacement de l'en-tête !
mainHdr.parentNode.replaceChild(mainHdr, header);

// D'ailleurs, on ne dit jamais assez bonjour... x2 !
header.parentNode.insertBefore(header.cloneNode(true), header);

// En revanche, ceci donnerait un h1 vide...
header.cloneNode(false);
// ... car on n'a pas cloné le nœud texte fils de header !
```

Attention tout de même à faire les choses plus directement dans votre code de production ! Le script ci-dessus se résumerait à ceci, s'il ne prenait pas tant de détours :

Listing 3-3 La version minimale du script précédent

```
var header = document.createElement('h1');
header.appendChild(document.createTextNode('Bonjour'));

var mainHdr = document.getElementById('main');
mainHdr.parentNode.replaceChild(mainHdr, header);
header.parentNode.insertBefore(header.cloneNode(true), header);
```

Element

L'interface `Element` étend les capacités de `Node` pour les nœuds représentant des éléments, c'est-à-dire des balises. Ces nœuds se distinguent principalement des autres en ce qu'ils peuvent avoir des attributs et des éléments fils[1].

Plutôt que de devoir créer à la main des nœuds `Attr` et les gérer individuellement, en plus de les ajouter ou les retirer à la liste `attributes` de l'élément, on dispose donc de méthodes dédiées, qui facilitent le travail. On verra plus tard, avec le DOM HTML, que même ces méthodes sont peu utilisées : on passe généralement par des propriétés spécifiques pour chaque attribut concret. Néanmoins, pour pouvoir réaliser un traitement générique, il faut les connaître.

1. D'accord, Document peut aussi avoir des éléments fils... Si on ne peut plus simplifier à bon escient...

Tableau 3–7 Propriétés et méthodes principales de l'interface Element

Propriété/méthode	Description
`tagName`	La seule propriété significative. Fournit le nom de la balise pour l'élément. En mode HTML, utilise la forme canonique (majuscule), en mode XML et autres modes sensibles à la casse, utilise la casse employée à la création. La propriété `nodeName` devient synonyme de celle-ci.
`getAttribute(name)`	Renvoie la valeur de l'attribut indiqué, ou la chaîne vide (`''`) si l'attribut n'existe pas. Prend en compte une éventuelle valeur par défaut.
`getElementsByTagName(tag)`	Tiens ! On l'avait déjà au niveau de `Document`, celle-ci. Mais justement : appelée sur un élément, elle restreint son champ de recherche au fragment du document situé à l'intérieur de cet élément. Potentiellement beaucoup plus performant, et souvent plus utile que la version globale.
`hasAttribute(name)`	Précise si l'attribut indiqué est présent (ou dispose d'une valeur par défaut, ce qui revient au même). Plutôt utile, car `getAttribute` ne distingue pas entre attribut à valeur vide et attribut absent. Attention : n'est pas pris en charge par MSIE ! Utilisez alors plutôt l'opérateur `in`.
`removeAttribute(name)`	Retire un attribut de l'élément. Attention : si l'attribut de ce nom dispose d'une valeur par défaut, une nouvelle définition d'attribut avec cette valeur prend la place de la définition retirée. Il n'est donc pas possible de retirer une valeur par défaut. Si l'attribut n'existait pas, n'a aucun effet.

En somme, rien de bien révolutionnaire par rapport à ce que `Node` permettait de faire, surtout dans la mesure où, dans la majorité des codes concrets, on utilisera des propriétés spécifiques, issues du DOM HTML, pour les attributs qui nous intéressent.

Text

C'est le type des nœuds représentant un fragment de texte. En simplifiant un peu (au mépris de quelques cas particuliers plutôt rares), pour un document HTML, un fragment de texte démarre au premier caractère différent de < après une balise, et se termine au premier caractère <. Il comporte donc toute la mise en forme du code source HTML : retours chariot, indentation (par espaces ou tabulations), etc. C'est pourquoi on a tant de nœuds texte « inutiles » au milieu de nos arbres DOM.

Techniquement, l'interface n'hérite pas immédiatement de `Node` : elle dérive de `CharacterData`, qui elle dérive de `Node`. On dispose donc des méthodes de `CharacterData`, qui permettent de modifier la valeur in situ, en insérant du texte au milieu de l'existant, en en supprimant ou remplaçant une portion, voire en remplaçant tout.

Ces méthodes, `appendData`, `insertData`, `deleteData` et `replaceData`, sont assez rarement utilisées, car elles vont au-delà des besoins courants. Même la propriété

officielle pour le texte (`data`), n'est pratiquement jamais utilisée nommément, la propriété générique `nodeValue` étant ici synonyme.

En revanche, on utilise trop peu la propriété `length`, qui contient la taille précalculée du contenu texte. On passe trop souvent par la méthode `length()` des objets `String`. Ainsi :

```
header.firstChild.length
```

est équivalente, mais potentiellement plus efficace (au niveau de la milli-seconde, entendons-nous bien...) que :

```
header.firstChild.nodeValue.length()
```

Les quelques méthodes restantes, `splitText`, et les nouveautés de niveau 3, `wholeText` et `replaceWholeText`, correspondent à des usages pour l'instant si rares qu'on ne fera pas plus que les mentionner.

NodeList et NamedNodeMap

Ce sont les deux principales interfaces pour manipuler des collections, c'est-à-dire, pour simplifier des listes de nœuds. La différence entre les deux est que `NodeList` utilise uniquement des positions (1er élément, 4e élément...) tandis que `NamedNodeMap` associe un nom à chaque nœud.

`NodeList` est très fréquemment utilisée, car c'est le type de `childNodes` et le type de retour de `getElementsByTagName`. Sa définition est triviale.

Tableau 3–8 Propriété et méthode de l'interface NodeList

Propriété/Méthode	Description
`item(index)`	Accède au nœud (interface `Node`) concerné. Il peut s'agir de n'importe quel type de nœud, bien entendu. Le premier nœud est en position `0` (zéro), le dernier en position `length - 1`. Toute position supérieure ou égale à `length` renverra `null`.
`length`	Taille de la liste.

Attention ! Certains scripts utilisent la notation `maListe[index]` au lieu de `maListe.item(index)`. Cela ne fait pas partie du standard et n'est donc pas portable.

`NamedNodeMap` est principalement utilisée comme type de la propriété `attributes`. On ne s'en sert donc que dans le cadre de traitements génériques. Sa définition est des plus simples, elle aussi .

Tableau 3–9 Principales propriétés et méthodes de l'interface NamedNodeMap

Propriété/Méthode	Description
`getNamedItem(name)`	Renvoie le nœud associé au nom passé, ou `null` si aucun nœud n'est associé à `name`.
`item(index)`	Accède au nœud (interface `Node`) concerné. Il peut s'agir de n'importe quel type de nœud, bien entendu. Le premier nœud est en position 0 (zéro), le dernier en position `length - 1`. Toute position supérieure ou égale à `length` renverra `null`.
`length`	Taille de la liste.
`removeNamedItem(name)`	Retire l'association basée sur le nom passé. Dans la mesure où `NamedNodeMap` n'est utilisée que pour `attributes`, le fonctionnement est similaire à celui de `removeAttribute` : si une valeur par défaut est définie, un nouveau nœud avec cette valeur est créé en remplacement du nœud retiré. Attention toutefois en cas de nom inexistant (pas d'association) : lève une exception NOT_FOUND_ERR.
`setNamedItem(node)`	Stocke le nœud comme valeur associée à `node.nodeName`. On évite donc d'y stocker plusieurs nœuds texte, par exemple, puisqu'ils ont tous le même nom (`#text`). Ceci dit, encore une fois, les seules `NamedNodeMaps` qu'on emploie en général sont les propriétés `attributes`et on les utilise principalement en consultation. Si on devait les modifier, on passerait des nœuds `Attr`, qu'on ne détaille pas ici.

À titre d'exemple, voici une petite fonction qui obtient en quelque sorte le `innerText` d'un nœud (élément ou texte), en concaténant récursivement les valeurs de tous ses nœuds texte internes.

Listing 3-4 Un exemple de parcours du DOM et d'utilisation de NodeList

```
function getInnerText(node) {
    var result = '';
    if (Node.TEXT_NODE == node.nodeType)
        return node.nodeValue;
    if (Node.ELEMENT_NODE != node.nodeType)
        return '';
    for (var index = 0; index < node.childNodes.length; ++index)
        result += getInnerText(node.childNodes.item(index));
    return result;
} // getInnerText
```

Attention, pour que cet exemple fonctionne sur MSIE, il faudra définir `Node` et ses constantes. Vous trouverez dans l'archive des codes source pour cet ouvrage (disponible sur le site des éditions Eyrolles) une démonstration complète de cette fonction.

DOMImplementation

Cette interface, accessible par la propriété `implementation` de l'objet global `document`, représente l'état de la prise en charge du DOM par le navigateur. Elle est utile pour vérifier qu'une fonctionnalité DOM est présente ou non.

De nombreux exemples et didacticiels sur Internet testent plutôt qu'une fonctionnalité est présente en vérifiant si une fonction spécifique existe. Par exemple, pour vérifier qu'on dispose du DOM niveau 2, de nombreux scripts écrivent :

```
if (document.getElementById && document.createTextNode)
```

Ce n'est pas mal, mais cela n'est pas toujours possible pour toutes les fonctionnalités susceptibles de nous intéresser. La façon générique de procéder consiste à utiliser la méthode `hasFeature` de cette interface, la seule qui nous intéresse. Cette méthode prend deux arguments : le nom officiel de la fonctionnalité et le niveau de DOM souhaité (une fonctionnalité pouvant évoluer d'un niveau à l'autre). L'argument de niveau peut valoir la chaîne vide (`''`) voire `null`, auquel cas il n'est pas pris en compte.

Les fonctionnalités sont définies de façon dispersée dans les spécifications du DOM et évoluent à chaque niveau. Elles correspondent principalement à des modules, et on trouve de-ci de-là dans la spécification une mention comme « Une application DOM peut utiliser la méthode `DOMImplementation.hasFeature(feature, version)` avec comme arguments "XML" et "3.0" (respectivement) pour déterminer si ce module est supporté ou non par l'implémentation. »

Ainsi, pour vérifier qu'on a le niveau 2, on peut écrire :

```
if (document.implementation.hasFeature('Core', '2.0'))
```

D'accord, ce n'est pas plus court, mais primo, c'est valable pour tout test de fonctionnalité, et secundo, c'est tout de même plus lisible qu'une série de tests d'existence de méthodes.

Vous l'avez compris le paramètre `version` peut valoir `null`, `''`, `'1.0'`, `'2.0'` ou `'3.0'`. Les valeurs utiles pour le paramètre `feature` incluent.

Tableau 3–10 Identifiants de fonctionnalités pour hasFeature

Fonctionnalité	Description
`AS-READ, AS-EDIT, LS-AS`	Sous-modules de Abstract Schemas (version 3.0).
`Core`	Fonctionnalités noyau (versions 2.0 ou 3.0, la 1.0 est garantie...).
`CSS`	Sous-module de style (version 2.0).
`Events`	Gestion événementielle (versions 2.0 ou 3.0).
`HTML`	Module HTML (versions 1.0 et 2.0).

Tableau 3–10 Identifiants de fonctionnalités pour hasFeature (suite)

Fonctionnalité	Description
`LS`, `LS-Async`	Sous-modules de Load and Save (version 3.0).
`Range`	Sous-module de Traversal and Range (version 3.0).
`StyleSheets`	Sous-module de style (version 2.0).
`Traversal`	Sous-module de Traversal and Range (version 3.0).
`Validation`	Validation de conformité aux grammaires (version 3.0).
`Views`	Manipulation des représentations visuelles (version 2.0).
`ViewsAndFormatting`, `VisualViewsAndFormatting`	Sous-modules d'extension de Views (version 3.0).
`XML`	Extensions XML (version 3.0).
`XPath`	Prise en charge de XPath (et souvent de XSLT ; version 3.0).

HTMLDocument

Nous entrons maintenant dans l'univers merveilleux du DOM HTML. Pourquoi merveilleux ? Parce qu'il va considérablement nous simplifier l'accès aux attributs HTML, pour commencer. Comme je l'ai déjà signalé, nous n'utiliserons pratiquement jamais `attributes`, ni même `getAttribute` ou `setAttribute` ! Le DOM HTML définit une interface spécifique pour chaque balise HTML officielle, avec des propriétés pour chaque attribut, et aussi des méthodes dédiées. C'est une mine d'or pour vos scripts, aussi ajoutez dès à présent la spécification à vos marque-pages :

http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/html.html

Vous noterez que le DOM niveau 3 n'a pas touché au module HTML. On a donc ici une spécification stable, qui a subi pas moins de cinq révisions mais est solidifiée depuis près de quatre ans. Il n'est pas étonnant qu'elle soit bien prise en charge par la majorité des navigateurs.

Mais nous verrons quelques exemples de cela plutôt avec l'interface suivante, `HTMLElement`. Pour l'instant, examinons `HTMLDocument`, la spécialisation de `Document` dans le contexte des pages web.

Au risque d'en choquer certains, j'affirme pour commencer que `HTMLDocument` n'ajoute rien de véritablement utile à `Document`. Sur les 11 propriétés et 5 méthodes, je ne trouve une réelle utilité qu'aux propriétés `title` et `body`. C'est tout. Permettez-moi tout de même de justifier cette opinion :

- `referrer` est bien plus utile côté serveur ; côté client, elle n'ouvre la voie qu'à des manipulations tordues de l'historique ou à du *tracking* de navigation de l'utilisateur, deux utilisations un peu douteuses...

- `domain` et `URL` me semblent inutiles la majorité du temps : avez-vous besoin qu'on vous dise sur quelle page vous êtes ? Et `domain` est trop souvent utilisé pour contourner le modèle de sécurité de JavaScript.
- Les collections `images`, `applets`, `links`, `forms` et `anchors` reviennent à de simples appels à `getElementsByTagName` (ou peu s'en faut).
- `cookie`, comme `referrer`, devrait plutôt être traité par la couche serveur, sans parler de la complexité inutile de sa manipulation en texte.
- `open()`, `close()`, `write(…)` et `writeln(…)` appartiennent au crétacé du Web et ne devraient jamais être utilisées maintenant qu'on a le DOM et, souvent, `innerHTML`.
- `getElementsByName` a perdu 99 % de son intérêt avec XHTML ; elle est avantageusement remplacée par un ou plusieurs appels ciblés à `getElementById`.

Voilà ! Du coup, il ne nous reste que deux propriétés qui me semblent utiles dans des scripts modernes.

Tableau 3–11 Les deux propriétés vraiment utiles de HTMLDocument

Propriété	Description
`body`	Référence directe sur l'élément <body> du document. Il est fréquent d'avoir besoin de cet élément (pour parcourir son contenu ou ajouter un fragment en début ou fin de document), ce qui nous économise un `getElementsByTagName('body').item(0)`, ce qui n'est pas rien…
`title`	Le titre de la page, en lecture/écriture. De nombreux scripts ajustent le titre pour refléter un état, ce qui permet de le signifier partout où le système d'exploitation reprend ce titre : barre de titre de la fenêtre, titre du bouton dans la barre des tâches, etc.

Si vous examinez l'interface `HTMLDocument` dans la spécification, vous remarquerez que les propriétés collections utilisent comme type `HTMLCollection` plutôt que `NodeList`. L'utilisation est strictement compatible (propriété `length` et méthode `item`, si si), mais `HTMLCollection` a en plus une méthode `namedItem`, qui prend un ID d'élément et renvoie l'élément dans la collection dont l'attribut `id` a cette valeur, ou `null` en cas d'échec. Une sorte de `getElementById` localisé.

HTMLElement

Enfin, voici l'interface qui spécialise `Element` pour les éléments de pages web. Elle est elle-même spécialisée par une interface pour chaque balise HTML officielle : on trouve par exemple des interfaces `HTMLFormElement`, `HTMLInputElement`, `HTMLHeadingElement`, etc. Prenez le temps d'aller les découvrir dans la spécification, pour utiliser leurs nombreuses propriétés dédiées, qui simplifient grandement le code.

L'interface `HTMLElement` définit simplement cinq propriétés communes, qui correspondent à ce que la DTD de HTML appelle les « attributs noyau ». Voici ces cinq propriétes, dont on se sert tout le temps :

Tableau 3–12 Propriétés communes à tous les éléments HTML

Propriété	Description
`className`	Équivalent de l'attribut HTML `class`. Ne peut s'appeler `class`, parce que ce dernier est un mot réservé en JavaScript. Contient donc un ou plusieurs noms de classes, séparés par des espaces.
`dir`	Équivalent de l'attribut HTML `dir`. Peu utilisée dans des pages à langue unique, ou en tout cas ne mélangeant pas langues occidentales et orientales : indique la direction (de gauche à droite ou de droite à gauche) du texte dans l'élément.
`id`	Ai-je vraiment besoin de vous expliquer celle-ci ? Je rappelle qu'un ID est unique dans tout le document.
`lang`	Équivalent de l'attribut HTML `lang`. Spécifie donc la langue du texte contenu, via un code RFC 1766 (comme. `fr-FR`).
`title`	Équivalent de l'attribut HTML `title`. Fournit donc le texte alternatif de l'élément, qui apparaît généralement dans une infobulle lorsqu'on laisse le curseur de la souris un bref moment sur l'élément.

Eh bien voilà ! Nous avons vu l'essentiel des interfaces du DOM et du DOM HTML. Le reste est à examiner dans la spécification (encore une fois, si le format de cette dernière vous égare, jetez donc d'abord un œil à l'annexe C).

Nous réaliserons quelques exemples concrets et parfois copieux plus loin dans ce chapitre, axés autour de besoins fréquents.

Quelques bonnes habitudes

Lorsqu'on travaille avec le DOM, c'est comme pour tout : il y a quelques bonnes façons de faire et beaucoup de mauvaises. Les conseils de cette section devraient déjà vous éviter une bonne partie des écueils et problèmes de maintenance.

Détecter le niveau de DOM disponible

Comme on l'a vu en parlant de l'interface `DOMImplementation`, de nombreux scripts détectent qu'ils disposent du DOM niveau 2 avec un code du style :

```
if (document.getElementById && document.createTextNode)
```

En effet, vous remarquez qu'on n'a pas ajouté les parenthèses d'appel derrière les noms des méthodes : on récupère donc juste les objets `Function` correspondants. Si ces méthodes existent, ces objets seront différents de `null`, et donc, traités comme des booléens, équivalents à `true`. La condition sera donc validée.

Vous vous demandez peut-être pourquoi on teste deux méthodes, alors que `getElementById` suffirait (elle a été introduite au niveau 2). C'est parce que certains navigateurs, dans d'anciennes versions, ne fournissaient qu'une prise en charge très partielle du DOM niveau 2, qui proposait généralement `getElementById` en raison de son immense popularité, mais plus rarement des fonctions comme `createTextNode`. Il s'agit donc d'une tentative vague de protection contre une prise en charge trop partielle.

C'est généralement fiable, mais ce n'est pas forcément extensible à tous les besoins. Il peut arriver qu'un module DOM, à un niveau précis, n'introduise aucune nouvelle propriété ou méthode (il peut se contenter de modifier le comportement de méthodes ou propriétés, en ajoutant des cas d'erreur, en limitant leurs résultats, ou en activant la sensibilité à la casse par exemple). Que faire alors ?

On l'a vu, la solution réside dans l'emploi de la méthode `hasFeature` de `DOMImplementation`. Comment récupérer un objet proposant cette interface ? C'est très simple : une implémentation conforme de `Document` doit fournir une propriété `implementation`, qui sert précisément à cela. Quant à l'interface `Document`, on a vu qu'elle était proposée par l'objet global JavaScript `document`. Et cela fonctionne en effet sur tous les principaux navigateurs.

Pour détecter un niveau général de DOM, on utilisera la fonctionnalité `Core`. On écrira donc pour détecter le support de DOM niveau 2 ou ultérieur :

```
if (document.implementation.hasFeature('Core', '2.0'))
```

Pour détecter la prise en charge de DOM niveau 3 XPath :

```
if (document.implementation.hasFeature('XPath', '3.0'))
```

C'est simple, et surtout lisible et explicite !

Créer les nœuds dans le bon ordre

Lorsqu'on crée un fragment DOM, c'est-à-dire une série de nœuds qu'on va imbriquer les uns dans les autres (ce qui peut être aussi bête qu'un élément avec un texte à l'intérieur !), on doit garder deux considérations à l'esprit :

- Intuitivement, les éléments doivent être crées de l'extérieur vers l'intérieur : on va créer l'élément avant de créer le nœud texte à placer dedans, par exemple. On va créer la liste avant de créer ses éléments, etc.
- En revanche, chaque fois qu'un nœud est ajouté dans un autre, et que ce nœud conteneur est attaché au DOM, le navigateur est susceptible de déclencher immédiatement un *reflow*, c'est-à-dire une mise à jour de l'affichage. Cette mise à jour entraînera peut-être des décalages tous azimuts. Par exemple, lorsqu'une ligne devient plus longue que la largeur disponible, le paragraphe auquel elle appartient occupe tout à coup une ligne supplémentaire, ce qui décale le reste...

Il est bon de suivre le premier point et de créer les éléments de l'extérieur vers l'intérieur. En revanche, pour composer les éléments ensemble, généralement à coup d'appels à `appendChild`, plus rarement à `insertBefore`, il est préférable d'attendre que le fragment soit complet pour ajouter l'élément racine du fragment (l'élément externe, si vous préférez) au DOM de la page.

De cette façon, on ne causera qu'un seul reflow, évitant ainsi d'éventuels impacts visuels disgracieux au fil de la construction du fragment. Qui plus est, plusieurs reflows auraient un impact négatif sur la vitesse d'exécution du script. En les évitant, on gagne donc tant visuellement qu'en rapidité !

Ne scripter qu'après que le DOM voulu soit construit

On touche ici à une source commune d'erreur chez les débutants qui scriptent le DOM. Il existe une règle simple, et même évidente : tant que le navigateur n'a pas lu et interprété un fragment donné de votre HTML, celui-ci n'est pas présent dans le DOM de la page.

Corollaire nécessaire : un script situé avant un élément dans le source HTML ne pourra pas immédiatement accéder à cet élément via le DOM. Dans la mesure où la salutaire séparation du contenu et du comportement impose d'utiliser uniquement des scripts externes, liés à la page à l'aide d'une balise du type :

```
<script type="text/javascript" src="chemin/fichier.js"></script>
```

et que ces balises sont par convention placées dans l'élément `<head>`, situé avant l'élément `<body>`, on a donc virtuellement la garantie que nos scripts seront analysés avant qu'il existe le moindre DOM pour le corps du document.

Par conséquent, un script qui contiendrait au niveau racine (c'est-à-dire hors de toute fonction) un code du style :

```
var header = document.getElementById('main-title');
```

est voué à l'échec : le HTML correspondant, probablement quelque chose du style `<h1 id="main-title">…</h1>`, n'a pas encore été traité à ce moment-là. Le navigateur traite en effet votre script au moment où il le charge, c'est-à-dire en traitant l'élément `<script>` qui l'invoque.

La solution est simple : faire une fonction pour votre code d'initialisation, et demander au navigateur d'appeler cette fonction une fois le DOM chargé. Par définition, à ce moment-là, l'ensemble du document aura été chargé, et sera donc représenté dans le DOM.

Idéalement, il faut réagir dès que le DOM est chargé, ce qui survient bien avant que la page elle-même ne le soit : entre les deux, le navigateur doit charger toutes les ressources de la page : CSS, images, applets, etc. Ce chargement complémentaire est potentiellement très long et cela peut retarder d'autant l'exécution de vos scripts d'initialisation. Hélas, certains navigateurs ne fournissent un événement que pour ce dernier chargement, le plus tardif.

Ce chargement « portable » correspond à l'événement `onload` de l'objet global `window`. Suivant ce que vous avez sous la main pour écrire votre script, y attacher votre fonction sera plus ou moins compliqué. Nous verrons tout à l'heure les détails de la gestion événementielle, mais sachez que cela n'a rien de foncièrement difficile. Si vous disposez de la bibliothèque Prototype par exemple (que nous verrons en détail au chapitre suivant), cela revient simplement à écrire :

```
Event.observe(window, 'load', myInitFunc, false);
```

Si vous n'avez pas Prototype mais avez la garantie d'un navigateur conforme au standard DOM niveau 2 événements (ce qui n'est pas le cas de MSIE, déjà...), c'est du même ordre :

```
window.addEventListener('load', myInitFunc, false);
```

Encore une fois, un peu de patience : nous verrons les détails un peu plus tard dans ce chapitre.

Pour ceux qui ont un besoin impérieux d'exécuter du script dès le DOM chargé, sans attendre le chargement de la page, il existe effectivement certains mécanismes, bien que rien encore ne soit véritablement standardisé. Une solution portable, qui étend la bibliothèque Prototype, est détaillée ici :
http://agileweb.org/articles/2006/07/28/onload-final-update.

Ne jamais utiliser d'extension propriétaire

Le Web est rempli d'articles, didacticiels et démonstrations écrits par des personnes qui n'ont pas forcément eu à cœur de s'en tenir aux standards. Souvent, « ça a marché pour eux, donc ça doit marcher pour vous ». Le problème, c'est que suite à la guerre des navigateurs des années 1990, chaque navigateur a mis en place des dizaines d'extensions propriétaires un peu partout dans JavaScript et le DOM, alors au fameux niveau zéro.

Lorsque vous tombez sur un script qui semble résoudre votre problème, commencez par vérifier scrupuleusement que toute portion technique dont vous ne maîtrisez pas le contenu (nom de propriété ou de méthode inconnue, etc.) est en réalité conforme aux spécifications. Il vous suffit de jeter un œil au standard de JavaScript et aux spécifications du DOM pour être fixé(e).

Bien sûr, on n'a parfois pas le choix : ainsi, lorsqu'une fonction standard n'est pas prise en charge par un navigateur, ni simulable en combinant d'autres fonctions standards, il faut bien recourir à la fonction propriétaire équivalente. Mais ces cas sont assez rares et correspondent presque toujours à une implémentation partielle ou incorrecte du DOM, généralement par MSIE.

Évitez par exemple d'utiliser `document.layers`, la fonction `GetObject`, la classe `ActiveXObject`, les commentaires conditionnels, etc. Souvenez-vous : le Web sera standard, ou ne sera plus !

Utiliser un inspecteur DOM

Il est temps de parler des outils. Nous l'avons déjà vu au chapitre 2, un bon outil d'aide au développement peut vous faire économiser un temps précieux. Lorsqu'il s'agit par exemple de déboguer du JavaScript, on peut choisir de passer la journée à traquer un problème subtil à coup de `alert`, ou de recourir à un débogueur intégré et de faire du pas à pas, pour trouver le souci en quelques minutes.

Un inspecteur DOM sert à afficher tout ou partie du DOM d'un document, sous forme d'une arborescence dépliable de nœuds. Il est bien entendu très utile pour mettre au point un script censé parcourir le DOM d'un document, puisqu'il évite de travailler à l'aveugle ou de recourir à une pléthore d'appels à `alert`.

L'inspecteur DOM de Firefox/Mozilla

Nous avons vu au précédent chapitre le débogueur « officiel » de Mozilla, baptisé Venkman. Il est mis à disposition sous forme d'une extension. L'inspecteur DOM, en revanche, fait partie intégrante du navigateur. Pour l'obtenir, il vous faut procéder différemment suivant votre plate-forme.

- Sous Windows (et, je crois, Mac OS X), il faut l'avoir installé explicitement, ce qui est à mon sens une bêtise de la part de la fondation Mozilla. Si vous n'aviez pas opté pour une installation personnalisée puis coché l'option, vous en êtes quitte pour relancer l'installation après avoir fermé toutes vos fenêtres Firefox. Il vous faudra alors choisir le mode personnalisé, et cocher lorsqu'on vous le proposera l'option Inspecteur DOM. Soyez tranquille, vous n'aurez pas de doublon dans la liste des programmes installés ou sur le disque, et vous conserverez tout votre profil d'utilisation.
- Sous Linux, suivant votre distribution, il fera ou non partie du paquet `firefox`. À vous de voir si vous disposez, dans le menu Outils, d'une option Inspecteur DOM. Dans la négative, recherchez le paquet idoine et installez-le après avoir fermé toutes vos fenêtres. Sur Debian par exemple, il s'appelle `firefox-dom-inspector`.

Une fois disponible, l'inspecteur s'obtient depuis le menu Outils>Inspecteur DOM, ou en pressant Ctrl+Maj+I. Il inspecte par défaut la page en cours, mais vous pouvez modifier ce comportement. Voici son aspect général.

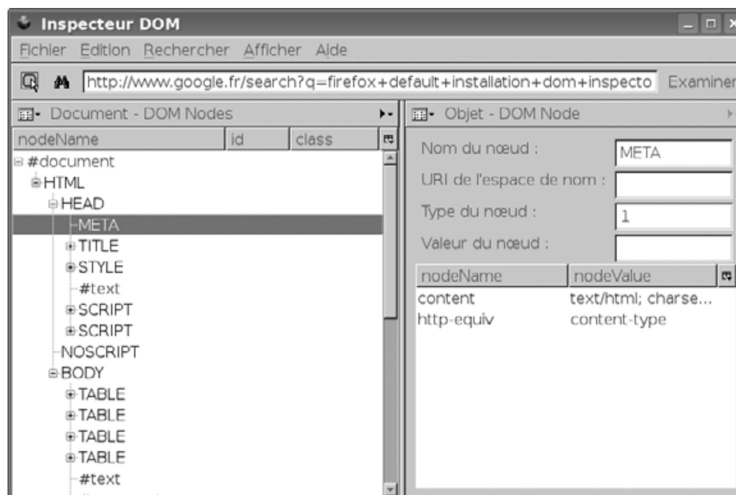


Figure 3–2
L'inspecteur DOM de Mozilla

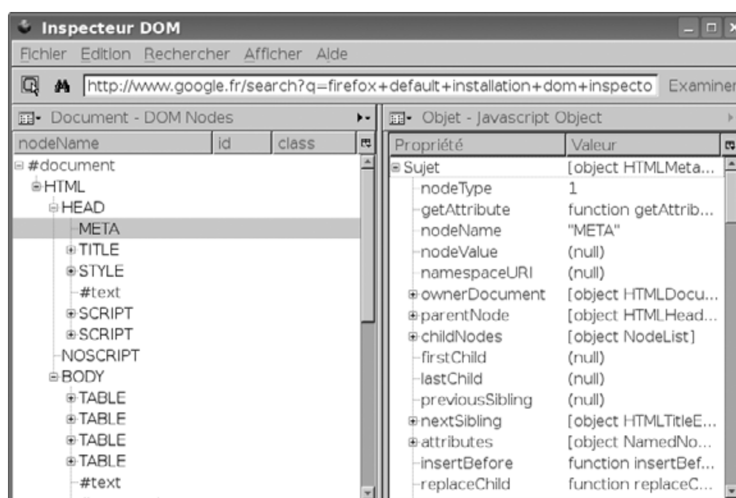
Sur la gauche, vous avez le DOM de la page, dépliable et navigable de façon classique. Sur la droite, vous avez, au choix, l'objet DOM pur (choix par défaut) ou l'objet DOM JavaScript correspondant au nœud sélectionné à gauche.

La figure 3-.2 montre le DOM d'une page de résultats Google, avec un nœud sélectionné qui propose plusieurs informations dans la vue DOM pur : le nom du nœud (ici sa balise canonique, puisqu'il s'agit d'un élément dans un document en mode HTML), son type (1, donc `Node.ELEMENT_NODE`), sa valeur (vide comme pour tout élément) et ses attributs.

On a parfois moins de chance, par exemple sur des nœuds texte, où tout ceci est remplacé par une vaste zone de texte, ce qui pour les nombreux nœuds texte vides n'est pas d'une grande utilité.

Prenons le panneau de droite. En haut à gauche de ce panneau, une icône permet, en cliquant dessus, d'afficher une liste déroulante, dans laquelle vous pouvez choisir l'option JavaScript Object. La liste qui s'affiche alors offre une vue bien plus détaillée du nœud courant ; c'est pratique pour aller dénicher l'information, mais cela peut aussi nous submerger. Par exemple, pour le même nœud que dans la figure précédente, on obtient ici, après avoir déroulé le premier élément de la liste, nommé « Sujet ».

Figure 3–3
La vue JavaScript Object pour le même nœud



C'est sans surprise : on a l'ensemble des propriétés et méthodes définies par la spécification du DOM. Vous avez peut-être remarqué qu'elles ne semblent pas être triées dans l'ordre alphabétique. Elles sont plus ou moins exactement groupées par niveau, module, interface et enfin ordre de définition dans la spécification. Ici, on a d'abord toutes celles de `Node` (`nodeType`, `nodeName`...), puis celles de `Element` (`attributes`, `getAttribute`, qui fait d'ailleurs exception ici en étant présent plus tôt), et plus bas, hors de la figure capturée ici, celles de `HTMLElement` (`id`, `title`...) et celles de `HTMLMetaElement` (`content`, `httpEquiv`). On trouve ensuite, les propriétés issues du module `Style` et les ajouts effectués à `Node` par le DOM niveau 3.

Voici un rapide tour des menus et services fournis :

- Le menu Fichier vous permet d'inspecter un autre document, soit en sélectionnant une des fenêtres ouvertes de votre navigateur, soit en précisant une nouvelle URL.
- Le menu Rechercher permet de chercher dans le DOM par nom de nœud, valeur d'ID ou valeur pour un attribut précis.

- Le menu Afficher permet de restreindre les types de nœuds affichés (en omettant les nœuds vides, par exemple), mais aussi de choisir si la vue navigateur met en exergue l'élément visuel sélectionné dans l'arborescence DOM (comportement par défaut) ou non. En effet, par défaut, sélectionner un élément dans l'arbre du DOM qui est affiché dans la page produit un cadre clignotant l'espace d'une seconde autour de cet élément, pour aider le développeur à s'y repérer.

L'inspecteur DOM de Firebug

Au chapitre précédent, en plus de Venkman, nous avons aussi vu les capacités de débogage JavaScript de l'extension Firebug pour Firefox. Nous vous avions promis que vous n'aviez pas fini d'entendre parler de cette extension ; le moment est venu de tenir cette promesse une première fois.

Firebug propose donc un inspecteur, et même plusieurs, accessibles en choisissant le bouton Inspect ou l'onglet Inspector :

- Un inspecteur *Source*, qui affiche un arbre des éléments avec leur représentation HTML (et une coloration syntaxique, s'il vous plaît !). Survoler un élément du document le sélectionne automatiquement.
- Un inspecteur *Style*, qui permet d'afficher les styles, explicites (attributs `style`) ou complets (c'est-à-dire incluant les valeurs par défaut et celles provenant de règles CSS), pour l'élément survolé. La bascule se trouve dans le menu Options de la barre Firebug.
- Un inspecteur *Layout*, qui affiche l'ensemble des propriétés relatives au positionnement pour l'élément survolé. Précieux quand on travaille sur du code à la script.aculo.us !
- Un inspecteur *Events*, qui affiche les événements lorsqu'ils se déclenchent.
- Enfin, un inspecteur *DOM*, qui peut fonctionner en mode global (affiche tous les objets globaux de la page et les propriétés de l'objet courant, donc `window`) ou en mode survol (affiche le DOM de l'objet survolé). On bascule de l'un à l'autre avec le bouton Inspect de la barre Firebug.

Les figures suivantes présentent quelques exemples d'aspect pour ces inspecteurs.

Vous trouverez de nombreux exemples supplémentaires (et impressionnants !) sur la page dédiée du site officiel : http://joehewitt.com/software/firebug/screens.php. Cliquez sur les vignettes pour voir les captures d'écran complètes.

Personnellement, j'utilise bien plus souvent l'inspecteur DOM de Firebug que celui de Firefox. Utilisez les deux quelques temps pour déterminer celui qui vous semble le plus agréable.



Figure 3–4
L'inspecteur Source avec le nœud title sélectionné

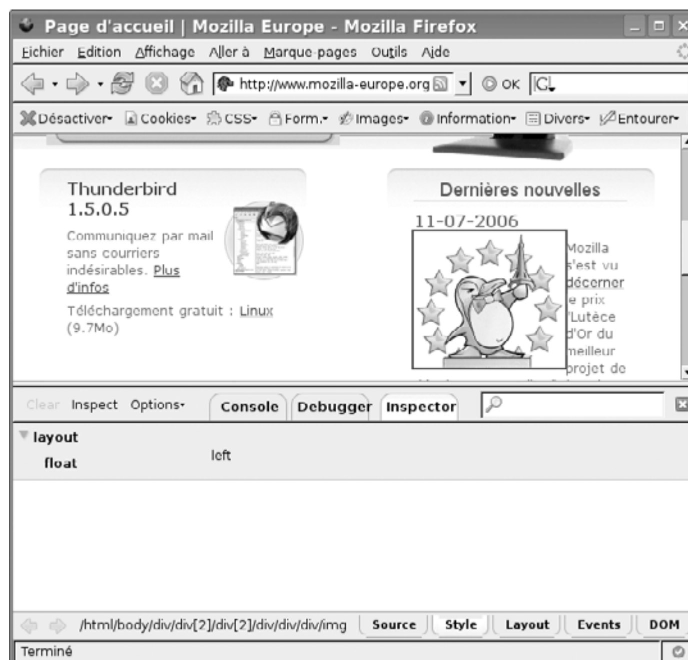


Figure 3–5
L'inspecteur Style au survol d'une image que son attribut style rend flottante

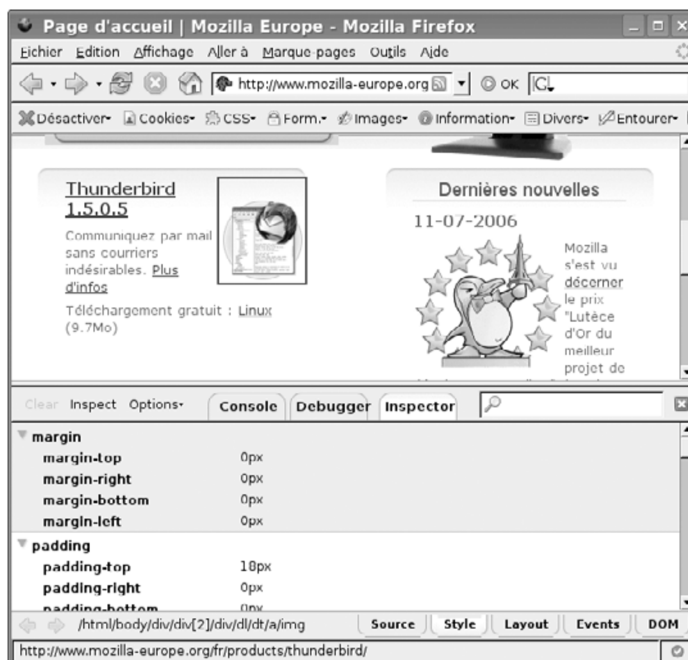


Figure 3–6
L'inspecteur Style en mode « styles complets » sur une autre image

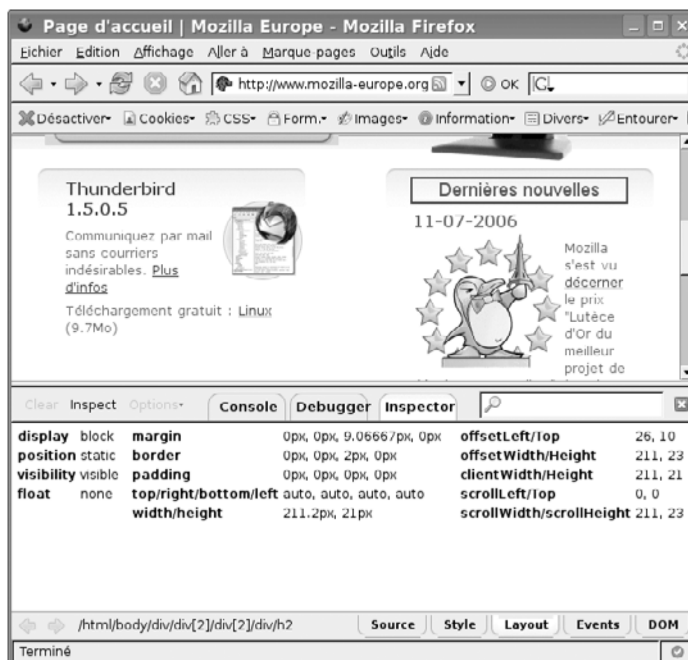


Figure 3–7
L'inspecteur Layout sur un titre : deux colonnes de styles et une de propriétés supplémentaires relatives au positionnement



Figure 3–8
L'inspecteur DOM en mode global



Figure 3–9
L'inspecteur DOM en mode survol, sur un lien

Répondre aux événements

Après ces quelques conseils, abordons le dernier point technique du DOM que nous n'avons pas vu : les événements. Sans gestion événementielle, c'est bien simple : votre page est morte. C'est-à-dire, pour faire une lapalissade, qu'elle n'est pas vivante. Elle ne réagit pas à la souris, au clavier, ni même aux événements internes ou basés sur des *timers*. La page est le bec dans l'eau.

Nous avons déjà évoqué brièvement la gestion événementielle. Ce sujet se découpe en deux grandes parties :

1 Associer un gestionnaire (une fonction que nous avons écrite) à un événement précis pour un objet précis.
2 Traiter cet événement lorsqu'il survient.

En raison de la fameuse guerre des navigateurs des années 1990, on a vu fleurir plusieurs manières totalement incompatibles entre elles de gérer des événements. Google n'oubliant jamais, on trouve encore de très nombreuses pages prônant l'une ou l'autre de ces méthodes aujourd'hui obsolètes, que ce soit pour associer l'événement à un objet ou pour traiter l'objet événement lorsqu'il survient.

Afin de vous aider à bien distinguer les principales approches et à comprendre en quoi toutes (sauf une !) posent problème, nous allons les étudier tour à tour. Il s'agit là de culture technique, car lorsque nous passerons à Prototype, au chapitre suivant, ces incompatibilités et ces détails techniques seront masqués.

Les truands : les attributs d'événement dans HTML

Mais si, vous les avez déjà vus : ce sont les attributs `on`*`xxx`* dans le HTML. Voici quelques exemples courants :

```
<body onload="initPage()">
…
<form method="post" action="process.php" onsubmit="checkForm()">
…
<a href="#" onclick="return popupWindow('help.html')">Aide</a>
…
<div onclick="doSomething()">
…
```

Certains de ces exemples cumulent les tares en enfreignant plusieurs règles techniques et éthiques... Tous posent en tout cas trois problèmes :

1 Ils représentent une intrusion du comportement dans le contenu.

2 Ils ne peuvent pas facilement associer plusieurs gestionnaires à un même événement pour un même élément.
3 Ils n'existent pas pour tous les événements de tous les éléments.

En somme, c'est à bannir, ne serait-ce que pour la première raison. La séparation stricte du contenu, de l'aspect et du comportement est un objectif permanent, une façon de travailler, un état d'esprit.

La brute : les propriétés d'événement dans le DOM niveau 0

« Très bien, » pensez-vous, « je vais déporter ces affectations dans mon JavaScript, au sein d'une fonction d'initialisation appelée après le chargement de la page, comme suggéré plus haut dans ce chapitre ».

L'idée est noble, et je vous en félicite avec un grand sourire, mais on peut tout de même mal l'exécuter. Par exemple, vous pourriez tomber dans le travers fréquent que voici :

Listing 3-5 Une association très imparfaite de gestionnaires d'événements

```
function initEventHandlers() {
    document.getElementById('mainForm').onsubmit = checkForm;
    document.getElementById('helpPopupLink').onclick = popupHelp;
    …
} // initEventHandlers

window.onload = initEventHandlers;
```

D'accord, vos gestionnaires d'événements sont désormais associés aux éléments dans le script, ce qui débarrasse votre HTML de tout attribut `on`*xxx*, n'y laissant plus que du contenu. Il y a effectivement du progrès.

Mais imaginez qu'après votre balise `<script>` chargeant ce fichier JavaScript, vous chargiez un script supplémentaire qui, lui aussi, a besoin de lancer une de ses fonctions au chargement. S'il procède de la même façon :

```
window.onload = myInitFunction;
```

Votre propre initialisation partira aux oubliettes !

On le voit, cette façon de procéder partage un inconvénient avec celle vue précédemment : il n'est pas possible d'associer plusieurs gestionnaires au même événement pour un même élément. Les bogues qui en résultent peuvent être très difficiles à diagnostiquer correctement.

Par ailleurs, là aussi, tous les événements potentiels ne disposent pas toujours d'une propriété idoine.

Le bon : addEventListener

Voici enfin la bonne façon de faire, qui est d'ailleurs celle spécifiée par le standard DOM Level 2 Events (www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/events.html).

Par définition, tout objet proposant l'interface `Node` constitue une cible d'événements potentielle et propose donc également l'interface `EventTarget`. Celle-ci fournit trois méthodes, dont deux nous intéressent particulièrement.

Tableau 3–13 Les deux méthodes clés de l'interface EventTarget

Méthode	Description
`addEventListener(type, listener, useCapture)`	Ajoute (inscrit, si vous préférez) un gestionnaire d'événement.
`removeEventListener(type, listener, useCapture)`	Retire (désinscrit) un gestionnaire d'événement.

Les deux méthodes ont les mêmes arguments, ne renvoient rien et ne lèvent aucune exception. Là où les deux premiers arguments sont simples à comprendre, le troisième est plus délicat.

- `type` décrit l'événement concerné : il s'agit d'un nom, généralement en minuscules, sans le préfixe `on`. La spécification précise la liste des événements valides du niveau 2, ainsi que les anciens événements du niveau 0, pour lesquels une compatibilité est maintenue jusqu'à présent. Un tableau est fourni plus bas.
- `listener` référence simplement la fonction de traitement.
- `useCapture` indique qu'on souhaite capturer l'événement plutôt que de l'intercepter lors de son bouillonnement. Ces notions un peu avancées sont présentées un peu plus loin dans ce chapitre. La plupart du temps, vous mettrez `false`.

Dans la majorité des cas, on ne prend pas la peine de désinscrire son gestionnaire, car celui-ci est valide pendant toute la durée de vie de la page (et puis, lorsqu'on utilise Prototype, ils sont automatiquement désinscrits quand on quitte la page). On utilise donc principalement `addEventListener`. Voici la version propre du listing précédent.

Listing 3-6 Une association de gestionnaires conforme aux standards

```
function initEventHandlers() {
    document.getElementById('mainForm').addEventListener(
      'submit', checkForm, false);
    document.getElementById('helpPopupLink').addEventListener(
      'click', popupHelp, false);
    ...
} // initEventHandlers

window.addEventListener('load', initEventHandlers, false);
```

Vous remarquerez que ce n'est guère plus compliqué, ni moins lisible. Avant d'entrer dans les détails sordides de la compatibilité avec MSIE et de cette histoire de capture et de bouillonnement, dressons une liste des principaux événements reconnus, décrits à la section 1.6 de la spécification :

Tableau 3–14 Principaux événements reconnus par le DOM niveau 2

Catégorie	Type	Description
UI (interface utilisateur)	`DOMFocusIn`	Équivalent de l'ancien `onFocus` : l'élément a reçu le focus clavier.
	`DOMFocusOut`	Équivalent de l'ancien `onBlur` : l'élément a perdu le focus clavier.
	`DOMActivate`	L'élément a été activé. La propriété `detail` de l'objet `Event` indique alors s'il s'agit d'une activation simple (1 : clic, touche `Entrée`) ou double (2 : double-clic, `Maj+Entrée`).
Souris	`click`	Clic de souris (ou équivalent clavier) : enfoncement puis relâchement d'un bouton.
	`mousedown`	Enfoncement d'un bouton.
	`mouseup`	Relâchement d'un bouton.
	`mouseover`	La souris commence à survoler l'élément (« entrée dans l'espace aérien »).
	`mousemove`	La souris survole l'élément.
	`mouseout`	La souris vient de cesser de survoler l'élément (« sortie de l'espace aérien »).
Clavier	Pas encore spécifiés, c'est ahurissant ! On se rabat pour l'instant sur les événements niveau 0, à savoir `keypress`, `keydown` et `keyup`.	

Chaque type d'événement utilise une version spécialisée de `Event` pour passer les détails au gestionnaire. Examinez la spécification pour obtenir la liste des propriétés spécifiques à chaque type.

Les événements compatibles DOM niveau 0 sont pour le moment toujours autorisés ; la spécification les nomme « événements HTML » ! En voici la liste.

Tableau 3–15 Événements de compatibilité avec le DOM niveau 0

Type	Description
`load`	Chargement terminé du document, de l'objet ou du cadre. Sans équivalent de niveau 2.
`unload`	La fenêtre ou le cadre va se fermer. À utiliser avec circonspection, peut s'avérer irritant ! Sans équivalent non plus (et c'est tant mieux !).
`abort`	Interruption volontaire du chargement du document, de l'objet ou du cadre.
`error`	Une image n'a pu se charger ou une erreur est survenue dans un script.
`select`	Du texte a été sélectionné dans le champ de saisie.
`change`	La valeur du champ a changé (déclenché à sa perte de focus uniquement, sauf, souvent, pour les `select`).
`submit`	L'utilisateur demande à envoyer le formulaire.

Tableau 3–15 Événements de compatibilité avec le DOM niveau 0 (suite)

Type	Description
`reset`	Le formulaire est réinitialisé (retour aux valeurs spécifiées dans le HTML).
`focus`	Le champ ou libellé (`label`, `input`, `select`, `textarea`, `button`) vient de récupérer le focus clavier.
`blur`	Le champ ou libellé a perdu le focus clavier.
`resize`	La fenêtre (ou en tout cas la vue contenant le document) est redimensionnée.
`scroll`	La fenêtre (...) subit un défilement.

Évitez toutefois de les utiliser lorsqu'un équivalent de niveau 2 existe, ne serait-ce que pour la pérennité de votre code...

Accommoder MSIE

Eh oui, on y revient toujours : alors que `addEventListener`, la méthode officielle, dont la spécification finale a déjà 6 ans, est prise en charge par l'ensemble des navigateurs répandus, MSIE n'en a jamais entendu parler. Non ! Dans MSIE, mes chers lecteurs, on utilise `attachEvent`. D'où cela sort-il ? De l'imagination féconde, quoique mal avisée, des développeurs du siècle dernier (littéralement). Remarquez, le nom a du sens, c'est déjà ça.

Cette méthode est décrite sur http://msdn.microsoft.com/workshop/author/dhtml/reference/methods/attachevent.asp, et si vous allez y jeter un œil, vous verrez qu'en dépit d'un certain culot consistant à nommer la liste propriétaire Microsoft des noms d'événements une « liste des événements DHTML *standards* », l'interface est simple : le nom d'événement, et la fonction de gestion. Pas de troisième argument en revanche, MSIE n'offrant pas de mécanisme de capture, mais uniquement le bouillonnement. Et les noms d'événements *standards* (quel humour, ces rédacteurs de documentation !) sont précédés du traditionnel préfixe `on`, comme au bon vieux temps.

La traduction est donc simple : dans MSIE, au lieu de faire :

```
node.addEventListener('event', handler, false);
```

On fait :

```
node.attachEvent('onevent', handler);
```

Rien de bien sorcier, mais si on pouvait éviter de coder le `if/else` à chaque association, ce serait mieux. On va donc écrire une fonction spécifique.

L'idéal, ce serait de la rendre disponible dans tous les objets, y compris tous les objets natifs du DOM, pour la manipuler de façon très similaire à `addEventListener`, mais hélas, c'est impossible : la plupart des objets natifs ne fournissent pas de prototype sur lequel greffer notre méthode. Il faut donc se contenter d'une bête fonction, à l'ancienne. Voici un exemple d'implémentation.

Listing 3-7 Un exemple de fonction portable d'association de gestionnaire

```
function addListener(element, baseName, handler) {
    if (element.addEventListener)
        element.addEventListener(baseName, handler, false);
    else if (element.attachEvent)
        element.attachEvent('on' + baseName, handler);
} // addListener
```

Notez que le cas restant (ni `addEventListener`, ni `attachEvent`) concerne si peu de navigateurs, tous un peu marginaux, qu'il ne mérite pas qu'on s'y intéresse, surtout qu'alors il n'est généralement pas possible de scripter la gestion événementielle !

J'ai mis cette fonction à titre d'exemple, et pour vous faire comprendre la mécanique. Dans la pratique, dès le prochain chapitre, on utilisera `Event.observe`, de Prototype, qui fait fondamentalement la même chose (avec tout de même plein de petits détails autour).

La propagation : capture ou bouillonnement ?

Depuis tout à l'heure, vous lisez partout capture et bouillonnement. Si vous n'êtes pas habitué aux mécanismes de propagation d'événements dans les navigateurs, ces termes vous sont étrangers (au moins un des deux).

Il s'agit des deux modes historiques d'interception des événements. Ils s'appliquent uniquement aux événements de l'interface utilisateur et non aux événements internes ou systèmes, comme le déclenchement d'un *timer*.

Il faut bien comprendre une chose : les événements sont traités par le navigateur indépendamment de l'existence de gestionnaires. Que vous ayez ou non défini des gestionnaires associés à un événement, lorsque celui-ci survient, le navigateur crée toujours un objet pour le représenter et notifie les éléments concernés que l'événement est survenu, en leur passant l'objet qui le représente.

Dans les explications qui vont suivre, on se basera sur le document HTML d'exemple de la figure 3-8.

Listing 3-8 Un document HTML simple pour nos explications

```
<html>
…
<body>
    …
    <div id="navbar">
        …
        <a id="homeLink" href="/home">Accueil</a>
        …
    </div>
</body>
</html>
```

Le modèle le plus courant : le bouillonnement

La plupart des événements bouillonnent. Pas tous ceci dit, vérifiez dans la spécification au cas par cas (c'est précisé pour chaque type d'événement). Par exemple, les événements de niveau 0 `load`, `unload`, `focus` et `blur` ne bouillonnent pas (ce qui est logique, si on y réfléchit après avoir lu cette section).

Un événement bouillonnant (ce qui ne signifie pas qu'il fera la une des journaux) est d'abord déclenché sur l'élément le plus proche d'après le contexte courant ; en clair, l'élément situé sous la souris (pour un événement souris), ou ayant le focus clavier (pour un événement clavier). Cet élément, qui constitue la cible physique de l'événement, est souvent appelé l'élément source. L'événement est ensuite déclenché à nouveau pour chaque nœud parent de l'élément source. L'événement remonte donc toute la hiérarchie du document, jusqu'au nœud racine `Document` lui-même. On dit que l'élément bouillonne, traduction un peu pataude pour le terme anglais *to bubble up*.

Ainsi, dans le document du listing 3-8, si l'utilisateur clique sur le lien `homeLink`, c'est d'abord ce lien qui va recevoir l'événement, puis son nœud parent, `navbar`, ensuite le `body`, l'élément `html`, et finalement `document`, l'objet global qui représente le nœud racine du DOM pour la page.

Ce qui est très important, c'est que chaque nœud sur ce chemin a l'opportunité de stopper la propagation, ou si vous préférez, d'interrompre le bouillonnement. La procédure standard pour cela consiste à appeler la méthode `stopPropagation()` de l'objet `Event` (et sous MSIE, c'est bien entendu différent : on met la propriété `cancelBubble` à `true`).

Pourquoi stopper la propagation ? C'est pratique lorsque vous savez que votre gestionnaire est censé être le « terminus » pour cet événement. Et c'est le cas la plupart du temps : les pages où un même événement doit être traité à plusieurs niveaux hiérarchiques sont rares.

La capture, ou comment jouer les censeurs

La capture n'est pas, contrairement à une idée répandue, simplement l'inverse du bouillonnement. En réalité, les deux mécanismes peuvent parfaitement coexister.

La capture est conçue pour permettre à un gestionnaire placé à un niveau donné du DOM de « censurer » l'événement qu'il reçoit pour tout le fragment dont son nœud est racine (en tout cas, au moment de l'association : un nœud descendant à ce moment-là, mais qui serait ensuite déplacé hors du fragment par une manipulation du DOM, serait toujours sujet à la capture).

Il ne s'agit donc pas de traiter l'événement à proprement parler, mais de le censurer d'après un algorithme correspondant à votre logique d'interface, que vous aurez implémentée dans le gestionnaire. Lorsque l'événement visé a lieu pour un élément descendant de celui sur lequel vous avez enregistré la capture (je dis bien descendant : pas le nœud lui-même, ni un nœud ailleurs dans le DOM), votre gestionnaire est déclenché. S'il stoppe la propagation (toujours avec la méthode `stopPropagation` de l'objet `Event`), l'événement ne sera pas déclenché sur son élément source, et ne bouillonnera donc pas le cas échéant.

Ainsi, prenons l'appel suivant :

Listing 3-9 Un exemple de capture pour censure inconditionnelle

```
document.getElementById('navbar').addEventListener('click',
    function(event) { event.preventDefault(); }, true);
```

Ce code empêcherait toute détection de clic par le contenu de `navbar`, et donc, entre autres, notre lien `homeLink`.

Il faut préciser que ce mécanisme est rarement utile. On s'en sert plus pour geler l'interface pendant un traitement (en refusant tout événement utilisateur jusqu'à nouvel ordre pour tout ou partie du document) que pour des besoins subtils et perfectionnés. Par ailleurs, certains événements ne sont pas capturables ; c'est le cas par exemple des événements relatifs au focus, de `mousemove` ou encore de `load` et `unload`.

Et pour finir, la cerise habituelle : le mécanisme de capture n'est pas pris en charge par MSIE. D'ailleurs, vous avez bien vu que `attachEvent` n'a pas de paramètre pour la capture, et vous chercherez en vain une méthode `captureEvent`, qui aurait trop ressemblé à son homonyme du Netscape de l'époque...

Pour résumer, faites simple : évitez la capture.

L'objet Event

Le déclenchement d'un événement donne lieu à la création d'un objet pour le représenter. Cet objet est censé implémenter l'interface `Event`, ainsi qu'une interface plus spécialisée selon le type de l'événement (par exemple, `UIEvent` ou `MouseEvent`). Je dis censé, car bien entendu, MSIE traite cela à sa façon.

Tout gestionnaire d'événement reçoit normalement l'objet `Event` en argument. Sous MSIE, il n'est pas passé en argument mais est présent dans l'objet global `window.event`.

Le module événements est l'un des points les plus sensibles de la faille entre MSIE et le respect du DOM niveau 2. Cette faille est bien sûr masquée par des bibliothèques comme Prototype. Afin de simplifier la description et de l'homogénéiser, les sections qui suivent présenteront toujours trois syntaxes pour chaque aspect :

- La syntaxe officielle du standard, nommée « DOM ». Elles supposent toutes que l'argument du gestionnaire s'appelle `event`.
- La syntaxe propriétaire de MSIE, seule supportée par ce dernier.
- La syntaxe portable offerte par Prototype, ce qui constitue certes un petit saut en avant vers le chapitre 4, mais vous rassurera à chaque fois quant à l'inutilité de devoir jongler manuellement entre les versions officielle et MSIE.

Récupérer l'élément déclencheur

Un gestionnaire a souvent besoin de récupérer l'élément source, ou cible, de l'événement déclenché. C'est particulièrement vrai quand vous déclarez un gestionnaire unique au niveau du conteneur pour traiter de façon similaire un même événement survenant chez plusieurs éléments descendants (si cet événement bouillonne, comme le fera par exemple un clic, votre gestionnaire sera forcément notifié).

- DOM : `event.target`
- MSIE : `window.event.srcElement`
- Prototype : `Event.element(event)`

Stopper la propagation

Nous avons déjà examiné ce mécanisme à la section sur la capture et le bouillonnement.

- DOM : `event.stopPropagation()`
- MSIE : `window.event.cancelBubble = true`
- Prototype : `Event.stop(event)` (annule aussi le traitement par défaut)

Annuler le traitement par défaut

La notion de traitement par défaut est très intéressante et critique pour de nombreuses utilisations.

La majorité des événements ont un traitement par défaut, qui correspond à ce que ferait le navigateur si vous ne définissiez aucun gestionnaire pour l'événement. Par exemple, cliquer sur un lien navigue vers la cible de ce lien ; soumettre un formulaire envoie les informations à la couche serveur.

Sauf demande explicite dans votre gestionnaire, ce traitement par défaut aura lieu. Or, une fonction de vérification de validité des saisies dans un formulaire voudra pouvoir empêcher l'envoi du formulaire, en plus de signaler les problèmes de saisie. De même, une fonction chargée d'afficher la cible d'un lien dans une fenêtre surgissante voudra empêcher la fenêtre contenant le lien de naviguer elle aussi vers la cible.

Vous trouverez de nombreux exemples sur le Web qui vous diront qu'il suffit que votre gestionnaire renvoie `false`, ce qui est aujourd'hui parfaitement périmé (au point que ça ne marche même plus dans MSIE !).

Les mécanismes opérationnels à ce jour sont les suivants :

- DOM : `event.preventDefault()`
- MSIE : `window.event.returnValue = false`
- Prototype : `Event.stop(event)` (stoppe aussi la propagation).

JavaScript, événements et accessibilité

Une page qui ne fonctionnerait qu'avec JavaScript activé, parce qu'elle reposerait intégralement sur JavaScript et les événements pour assurer son rôle, constituerait un grave problème d'accessibilité au sens large.

D'abord, de nombreux contextes n'auront pas JavaScript (navigateurs textuels, certains navigateurs sur périphériques mobiles : téléphones, Palm Pilot/Visor, etc.), auront JavaScript désactivé (par décision des responsables informatiques de l'entreprise), ou auront une prise en charge très partielle (lecteurs d'écran). Tous ces utilisateurs, pourtant parfaitement légitimes, ne pourront utiliser correctement la page.

Même avec JavaScript opérationnel, il est irresponsable d'exiger des manipulations clavier ou souris complexes de tous vos utilisateurs : certains souffrent peut-être d'un handicap moteur ou ont un périphérique (*touchpad* ou *trackpoint* sur un portable) les empêchant de manipuler la souris avec précision ; d'autres ne peuvent confortablement lui associer un modificateur clavier (par exemple, si vous attendez un Ctrl+Clic), voire tout simplement d'utiliser la souris (handicap moteur plus lourd). Utiliser des événements souris de façon exclusive est très vite limitatif.

Quand bien même vous décideriez d'envoyer paître la base utilisateur « handicapée » au sens large (handicaps moteurs, visuels, cognitifs), soit plus de 15 % des internautes mondiaux (plus d'un million de personnes rien qu'en France), vous n'aurez peut-être pas le choix : un niveau élevé d'accessibilité est aujourd'hui une exigence légale pour tout appel d'offres émanant du service public, et un nombre croissant d'appels d'offres privés l'exigent également.

Accessibilité ne rime pas avec impossibilité, ni même avec complexité : il s'agit simplement d'ajuster nos habitudes de développement pour l'intégrer dans nos réflexes de code. Vous trouverez une panoplie très complète de documentations pratiques en français traitant de l'accessibilité et le Web sur le site d'Accessiweb, la cellule spécialisée de l'association BrailleNet : http://www.accessiweb.org/.

Voici déjà quelques conseils à retenir :

- Une pierre d'angle est bien sûr l'*unobstrusive JavaScript*, déjà discuté, qui consiste à ne jamais mettre de scripts ou d'attributs événementiels dans votre HTML : n'y laissez que le contenu !
- Autre réflexe important : pour assurer que votre page se dégrade élégamment (c'est-à-dire continue de fonctionner correctement au fur et à mesure que les moyens du bord se restreignent : plus de CSS, plus d'image, plus de JavaScript...), le mieux est de la réaliser par amélioration progressive.
  En d'autres termes, commencez par faire une page capable de fonctionner sans aucun JavaScript, mais qui n'aura recours qu'à des allers-retours avec la couche serveur. Ensuite, améliorez-la par petites touches à coups d'ajouts de gestionnaires depuis votre fichier de script (toujours *unobstrusive*...). En partant du bas, vous garantissez que la page est capable d'y retourner !
- Les fonctions de confort réalisées en JavaScript n'ont pas obligatoirement à avoir un équivalent classique ; ce qui compte, c'est que la page puisse fonctionner, donc rendre le service qui est sa raison d'être, sans JavaScript. Tant pis si c'est alors, fatalement, un peu plus pénible, un peu plus ardu. L'exemple typique est le tri de listes : sans JavaScript, on doit faire un aller-retour à chaque déplacement d'un élément vers le haut ou vers le bas ; avec JavaScript, on peut glisser-déplacer tout ce beau monde vite fait et valider à la fin !

Malgré beaucoup d'astuce dans l'emploi de JavaScript, il reste toutefois difficile de rendre facile d'emploi, particulièrement au clavier, des interfaces dynamiques un peu riches, comme des arborescences dépliables et repliables, des menus déroulants à multiples niveaux ou encore des grilles de données.

Mais sur ce front, l'espoir renaît : pour faciliter l'accessibilité de ces réalisations, une initiative « DHTML accessible » a vu le jour conjointement entre le W3C, IBM et la fondation Mozilla. Elle est d'ores et déjà implémentée dans Firefox 1.5. À l'aide d'attributs supplémentaires décrivant le rôle fonctionnel des éléments, il est possible

de faciliter grandement l'utilisation et la navigation au clavier de composants visuels riches et complexes (principalement à l'aide des flèches et de Tabulation, comme sous Windows ou Mac OS X, par exemple). Cela ouvre des horizons ! Vous en trouverez davantage sur la page dédiée du site Mozilla Developer Connection (MDC) et sur les pages concernées du W3C :

- http://developer.mozilla.org/en/docs/Accessible_DHTML
- http://www.w3.org/WAI/PF/roadmap/

L'initiative se penche d'ailleurs aussi sur l'accessibilité d'Ajax. On ne peut que l'encourager !

Enfin, quelques techniques utiles :

- Les *accessible pop-ups*, ou fenêtre surgissantes accessibles, décrivent comment faire pour qu'un lien s'affiche dans une fenêtre surgissante si JavaScript est actif, ou suive sa navigation traditionnelle dans le cas contraire. C'est un grand classique, décrit avec tous les détails de mise au point dans http://www.alistapart.com/articles/popuplinks/. Pour vraiment se blinder, on vérifiera que la fenêtre a bien été ouverte (un bloqueur de *pop-ups* un peu trop zélé pourrait l'en avoir empêché), comme décrit sur http://cookiecrook.com/AIR/2003/train/xmp/popup/pop.js.
- Ne modifiez le focus par script qu'avec parcimonie. En effet, changer le focus sans intervention de l'utilisateur peut se révéler extrêmement gênant pour ceux utilisant une loupe d'écran voire un lecteur d'écran. C'est donc à éviter, tout particulièrement si vous pensiez le faire périodiquement (par exemple, toutes les 30 secondes) ! En revanche, c'est parfaitement acceptable suite à une action manuelle de l'utilisateur (par exemple, en réaction à l'activation du bouton Lire mes courriels, on peut déplacer le focus sur la liste des courriels après avoir chargée celle-ci).
- Ne limitez pas vos événements traités à ceux spécifiques à la souris, sauf peut-être pour `click`, simulé par tous les navigateurs avec la touche Entrée. Préférez coupler la version souris et la version clavier, par exemple `mouseover` avec `focus` et `mouseout` avec `blur`, etc.

Besoins fréquents et solutions concrètes

Armés de toutes ces connaissances nouvelles, nous allons à présent les mettre en application au travers de quelques exemples concrets, qui répondent par ailleurs à des besoins récurrents. Nous ne détaillerons pas chaque script, il s'agit plutôt de faire une démonstration générale. Utilisez la spécification pour éclaircir les éventuels détails que vous ne saisiriez pas bien. Tous ces exemples figurent dans l'archive des codes source disponible sur la page de l'ouvrage sur le site des éditions Eyrolles.

Précisons aussi, avant de commencer, que ces exemples seraient souvent considérablement simplifiés par l'emploi judicieux des améliorations fournies par Prototype, mais ne mettons pas la charrue avant les bœufs...

Décoration automatique de labels

Commençons par une décoration automatique de labels. Comme vous le savez, l'élément `label` sert à identifier le libellé d'un champ de formulaire. On peut l'associer au champ soit en affectant l'ID du champ à l'attribut `for` du label, soit en plaçant le champ dans l'élément `label`, après (ou avant) le texte du libellé.

Nous allons gérer une décoration automatique (au chargement de la page) de ces libellés centrée sur deux aspects :

1. Tout libellé disposant d'un attribut `accesskey` tentera de souligner la lettre correspondante dans son texte, pour rendre la touche de raccourci évidente visuellement.
2. Tout libellé disposant d'un attribut `for` (méthode préférée d'association) vérifiera que le champ existe, et si tel est le cas, examinera l'ID du champ pour y détecter le texte `'Req'` : si ce texte est présent, il considérera que le champ est requis, et s'ajoutera donc la classe CSS `required`, que nous aurons définie comme affichant le texte en gras (et sur les navigateurs supportant la pseudoclasse `:after` et la propriété `content`, nous ajouterons une astérisque dynamique après le libellé).

Voici le fichier `index.html` sur lequel nous allons appliquer notre script :

Listing 3-10 Le HTML qui va subir notre décoration automatique

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Exemple DOM n°1 : décoration automatique de labels
    </title>
    <link rel="stylesheet" type="text/css" href="demo.css" />
    <script type="text/javascript" src="demo.js"></script>
</head>
<body>
<h1>Décoration automatique de labels</h1>
<p>Examinez le code source du formulaire
ci-dessous, et comparez à ce que vous obtenez visuellement.</p>
```

```
<p>(Notez que l'envoi du formulaire ne mènera à rien de particulier)</p>

<form id="demoForm" method="get" action="http://www.example.com">
<p>
    <label for="edtReqLogin" accesskey="D">Identifiant</label>
    <input type="text" id="edtReqLogin" name="login" tabindex="1" />
</p>
<p>
    <label for="edtFirstName" accesskey="P">Prénom</label>
    <input type="text" id="edtFirstName" name="firstName" tabindex="2" />
</p>
<p>
    <label for="edtLastName" accesskey="N">Nom</label>
    <input type="text" id="edtLastName" name="lastName" tabindex="3" />
</p>
<p class="submit">
    <input type="submit" value="Envoyer" accesskey="E" tabindex="4" />
</p>
</form>

</body>
</html>
```

Notez l'absence de tout balisage non sémantique : ni `table` ni fatras de `div` et `span` pour la mise en page. Et pourtant, on pourra obtenir un aspect irréprochable grâce à la feuille de styles, dont voici le contenu (`demo.css`) :

Listing 3-11 Notre feuille de styles pour cet exemple

```
form#demoForm {
    width: 40ex;
    padding: 1em;
    margin: 2em auto;
    border: 1ex solid silver;
    background: #eee;
    font-family: sans-serif;
}

form#demoForm p {
    position: relative;
    margin: 0 0 0.5em;
}

form#demoForm p.submit {
    margin: 0;
    text-align: right;
}
```

```
input#edtReqLogin, input#edtFirstName, input#edtLastName {
    position: absolute;
    left: 20ex;
    right: 0;
}

input:focus {
    border: 2px solid black;
    background: #ffd;
}

label.required {
    font-weight: bold;
}

label.required:after {
    content: '*';
}

span.accessKey {
    text-decoration: underline;
}
```

Enfin, voici notre script `demo.js`, qui constitue le sel de l'exemple :

Listing 3-12 Notre script de décoration automatique des libellés

```
// Être compatible avec MSIE...
if ('undefined' == typeof Node)
    Node = { ELEMENT_NODE: 1, TEXT_NODE: 3 };

function addListener(element, baseName, handler) {
    if (element.addEventListener)
        element.addEventListener(baseName, handler, false);
    else if (element.attachEvent)
        element.attachEvent('on' + baseName, handler);
} // addListener

function decorateLabels() {
    var labels = document.getElementsByTagName('label');
    for (var index = 0; index < labels.length; ++index) {
        var label = labels[index];
        if (label.accessKey) {
            var ak = label.accessKey.toUpperCase();
            decorateNodeForAccessKey(label, ak);
        }
```

```
        if (label.htmlFor) {
            var elt = document.getElementById(label.htmlFor);
            if (!elt)
                continue;
            if (elt.id.match(/Req/))
                label.className += ' required';
        }
    }
} // decorateLabels

function decorateNodeForAccessKey(elt, key) {
    if (Node.ELEMENT_NODE == elt.nodeType) {
        var node = elt.firstChild;
        while (node && !decorateNodeForAccessKey(node, key))
            node = node.nextSibling;
        // Si node n'est pas null, on a trouvé l'AK dans un descendant
        // et on a décoré : on renvoie non-null, équivalent à true
        return node;
    }
    if (Node.TEXT_NODE != elt.nodeType)
        return false;
    var pos = elt.nodeValue.toUpperCase().indexOf(key);
    if (-1 == pos)
        return false;
    var suffix = elt.nodeValue.substring(pos + 1);
    var akSpan = document.createElement('span');
    akSpan.className = 'accessKey';
    akSpan.appendChild(document.createTextNode(elt.nodeValue.charAt(pos)));
    // On évite node.splitText et node.deleteData sur MSIE...
    // On manipule nodeValue et on crée le deuxième noeud Texte manuellement.
    elt.nodeValue = elt.nodeValue.substring(0, pos);
    elt.parentNode.appendChild(akSpan);
    elt.parentNode.appendChild(document.createTextNode(suffix));
    // Très important pour éviter une récursion infinie !
    return true;
} // decorateNodeForAccessKey

addListener(window, 'load', decorateLabels);
```

Voici notre page après chargement.



Figure 3–10
Notre page chargée, avec ses libellés décorés

C'est déjà un bel exemple d'application, plutôt complet. Je vous encourage à l'améliorer au travers de deux petits exercices.

1. Listez toutes les valeurs de l'attribut `accesskey` (pas seulement dans les labels, mais dans tous les éléments : utilisez `getElementsByTagName('*')`) et détectez les collisions : ajoutez alors un cadre rouge aux libellés à l'aide de leur propriété `style` : `label.style.border = '2px solid red';`. Vous vous rendrez ainsi un fier service, car on a vite fait d'associer deux fois le même raccourci au fil de l'évolution de la page.
2. Toujours en itérant sur tous les éléments dotés d'un attribut `accesskey`, ajustez l'attribut `title` des éléments bénéficiant du raccourci (pour un label, il faut aller sur l'élément référencé dans `for`), soit en lui ajoutant un texte de type `'(Alt+X)'` si une valeur existe déjà, soit en créant la valeur `'Alt+X'`. Si vous voulez pousser, vous pourrez même détecter que vous êtes sur Mac OS, en cherchant par exemple le texte `'Macintosh'` à l'intérieur de `navigator.userAgent` et utiliser `'Cmd'` plutôt que `'Alt'`...

Validation automatique de formulaires

Toujours plus fort, nous allons maintenant fournir une validation avancée de formulaires. Ce chapitre se concentrant sur le DOM, je ne vous fournis plus à présent que le script. Vous trouverez la démonstration complète dans l'archive des codes source disponible sur le site des éditions Eyrolles.

Qu'entendons-nous par « validation automatique » ?

1 Chaque formulaire du document se voit associer notre gestionnaire d'interception de l'événement `submit` (qui est cumulatif aux autres gestionnaires éventuellement enregistrés).
2 Le *submit* est donc intercepté par notre gestionnaire, qui récupère tous les champs (`input`, `select`, `textarea`) du formulaire et examine leurs ID. Nous prenons en charge une syntaxe particulière dans les ID, décrite plus bas, qui permet de spécifier les contraintes de validation.
3 On accumule au fur et à mesure le texte des messages d'erreur dans un unique message. On garde également une référence vers le premier champ fautif.
4 En fin de traitement, si aucune erreur n'a été détectée, on ne fait rien, ce qui laisse passer l'envoi du formulaire. En revanche, si on a détecté un pépin, on signale l'erreur (ici avec un `alert`, mais vous pourriez avoir un `div` exprès pour ça, créer une liste `ul/li` de toute pièce, etc.), on met le focus sur le premier champ, on stoppe la propagation et on annule le traitement par défaut.

La syntaxe que nous prendrons en charge pour les ID est la suivante :

`préfixeQuelconque_[Req][_(Int|Dbl|Date)][_min[_max]]`

Quelques détails d'interprétation :

- `Req` indique, comme tout à l'heure, que le champ est requis.
- `Int` indique un champ à valeur entière, `Dbl` un champ `Double`, donc à virgule flottante, et `Date` un champ date, pour lequel on exigera ici, par simplicité, un format `jj/mm/aaaa`. On ne validera pas la date en profondeur (libre à vous...).
- Pas de `max` sans `min` d'abord. On ne les prend pas en charge pour les dates non plus, par souci de simplicité de l'exemple.

Voici le fragment de HTML qui contient notre formulaire :

Listing 3-13 Notre formulaire et ses ID spécialement conçus

```
<form id="demoForm" method="get" action="http://www.example.com">
<p>
    <label for="edtLogin_Req" accesskey="D">Identifiant</label>
    <input type="text" id="edtLogin_Req" name="login" tabindex="1" />
</p>
```

```
<p>
    <label for="edtAge_Req_Int_1_120" accesskey="A">Age</label>
    <input type="text" id="edtAge_Req_Int_1_120" name="age"
           ➥ tabindex="2" value="toto" />
</p>
<p>
    <label for="edtEuroRate_Dbl_0.01" accesskey="T">Taux de l'euro
    </label>
    <input type="text" id="edtEuroRate_Dbl_0.01" name="euroRate"
           ➥ tabindex="3" value="6.55957" />
</p>
<p>
    <label for="edtBirthDate_Date" accesskey="N">Date de naissance
    </label>
    <input type="text" id="edtBirthDate_Date" name="birthDate"
           ➥ tabindex="3" value="04/11/1977" />
</p>
<p class="submit">
    <input type="submit" value="Envoyer" accesskey="E" tabindex="4" />
</p>
</form>
```

Et voici les fragments importants du script (le reste est sur le site…) :

Listing 3-14 Notre validation automatique de formulaires

```
REGEX_AUTO_FIELD = /^[^_]+(_Req)?(_(Int|Dbl|Date)(_[0-9.]+){0,2})?$/;
REGEX_BLANK = /^\s*$/;
REGEX_DAY = /^(0?[1-9]|[1-2][0-9]|3[01])$/;
REGEX_MONTH = /^(0?[1-9]|1[0-2])$/;
// Les multiples groupes vont nous découper l'ID tout seuls...
REGEX_TYPED_FIELD = /_(Int|Dbl|Date)(_([0-9.]+))?(_([0-9.]+))?$/;
REGEX_YEAR = /^[0-9]{2,4}$/;
…
function addFormChecks() {
    var forms = document.forms;
    for (var index = 0; index < forms.length; ++index) {
        var form = forms.item(index);
        addListener(form, 'submit', checkForm);
    }
} // addFormChecks
…
function checkForm(e) {
    // Compatibilité MSIE / les autres...
    e = e || window.event;
    var form = e.target || e.srcElement;
    var errors = '';
    var faulty = null;
    for (var index = 0; index < form.elements.length; ++index) {
        var field = form.elements.item(index);
```

```
        // Vérification de syntaxe
        if (!field.id.match(REGEX_AUTO_FIELD))
            continue;
        var value = getFieldValue(field);
        // Champ requis ?
        if (field.id.match(/_Req/) && value.match(REGEX_BLANK)) {
            errors += getFieldName(field) + MSG_BLANK + '\n';
            faulty = faulty || field;
            continue;
        }
        // Champ typé ?
        var match = field.id.match(REGEX_TYPED_FIELD);
        if (match) {
            var type = match[1];
            var min = match[3];
            var max = match[5];
            var error = checkTypedField(value, type, min, max);
            if (error) {
                errors += getFieldName(field) + error + '\n';
                faulty = faulty || field;
            }
        }
    }
    if (!faulty)
        return;
    stopEvent(e);
    alert(errors);
    faulty.focus();
} // checkForm

function checkTypedField(value, type, min, max) {
    // Valeurs par défaut pour les bornes
    min = min || Number.NEGATIVE_INFINITY;
    max = max || Number.POSITIVE_INFINITY;
    var val;
    if ('Int' == type) {
        try {
            val = parseInt(value, 10);
            if (String(val) != value)
                throw val;
        } catch (e) {
            return MSG_NOT_AN_INTEGER;
        }
    }
    if ('Dbl' == type) {
        try {
            val = parseFloat(value);
            if (String(val) != value)
                throw val;
        } catch (e) {
```

```
            return MSG_NOT_A_DOUBLE;
        }
    }
    if ('Int' == type || 'Dbl' == type) {
        if (val < min)
            return MSG_TOO_LOW;
        if (val > max)
            return MSG_TOO_HIGH;
    }
    if ('Date' == type) {
        var comps = value.split('/');
        if (3 != comps.length || !comps[0].match(REGEX_DAY) ||
            !comps[1].match(REGEX_MONTH) ||
            !comps[2].match(REGEX_YEAR))
            return MSG_NOT_A_DATE;
    }
    return null;
} // checkTypedField
…
```

Les fonctions de service (`getFieldName`, `getFieldValue`, `stopEvent`) et les textes des messages sont laissés de côté, car ils n'apportent pas grand-chose à la fonctionnalité pure de l'exemple. Vous les trouverez dans l'archive disponible en ligne. On a déjà un bon script d'exemple, qui illustre la majorité des éléments techniques vus dans ce chapitre et le précédent !

Je ne saurai trop vous recommander de vous faire la main à l'aide des exercices suivants :

1. Ajoutez un gestionnaire au chargement qui donne le focus à l'élément de `tabindex` valant 1 (un) dans la page.
2. Augmentez la richesse de la syntaxe, en permettant en cas de type `Date` de définir, avant les bornes mais séparé de `'Date'` par un souligné (_), le format de date, parmi les possibilités suivantes : `dmy`, `mdy` et `ymd`. Vous pourrez alors ajuster la validation. Encore mieux : faites qu'un `y` (minuscule) signifie « année sur deux chiffres » et un `Y` (majuscule) « année sur quatre chiffres ».

Vous pouvez voir un exemple d'exécution à la figure 3-11.

Résoudre les écueils classiques

Ce dernier exemple, en particulier dans sa version complète, disponible en ligne, illustre les principaux écueils du scripting DOM, qui tournent principalement autour des différences entre MSIE et les autres navigateurs.

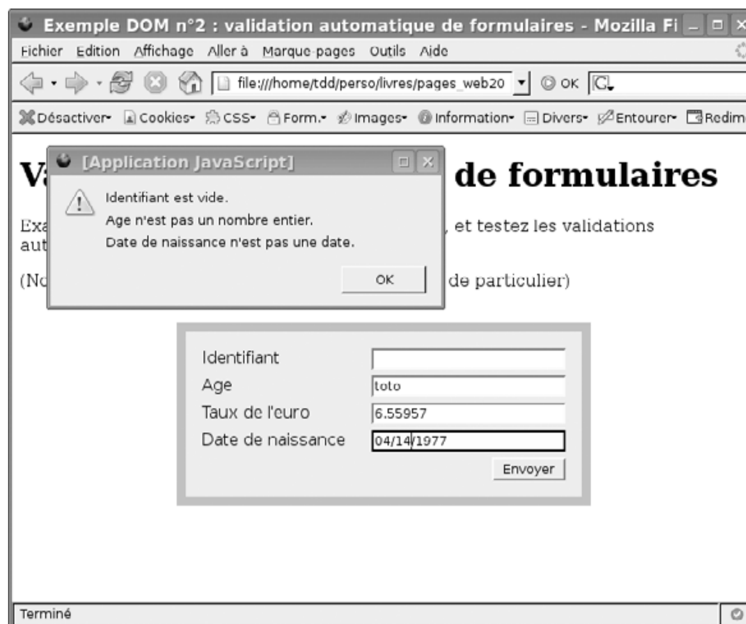


Figure 3–11
Exemple de validation automatique

Il existe aussi, bien sûr, des problèmes plus pointus, présents sur d'autres navigateurs répandus, comme des soucis de positionnement complexes ou de noms d'événements sur Firefox, Safari, etc. Mais il ne s'agit pas d'écueils classiques.

MSIE et la gestion événementielle

On l'a vu dans la section dédiée à ce sujet, quand il s'agit d'événements, MSIE ne fait rien comme les autres : pas de `stopPropagation()`, pas de `preventDefault()`, pas d'interface `Event`, de propriété `target`, ni d'objet événement passé en argument aux gestionnaires...

Vous avez pu voir des contournements manuels dans ce dernier script d'exemple, mais la solution la plus portable consiste à utiliser Prototype et son objet global `Event`. En plus d'offrir des méthodes équivalentes universelles (`stop`, `element`, etc.), il fournit une interface unifiée sur l'ensemble des propriétés spécifiques des événements (état des modificateurs clavier, position de la souris et j'en passe).

MSIE et le DOM de select/option

Il existe également un problème sur MSIE concernant le DOM des éléments `select` et `option`, qui décrivent les champs de type liste. Le DOM prévoit deux comportements :

- Une propriété `value` directement au niveau du `select`, qui fournit la valeur actuellement sélectionnée s'il y en a une (la première en cas de sélection multiple).
- Une propriété `value` au niveau des `options`, qui prend automatiquement la valeur de la propriété `text` si l'attribut `value` n'est pas défini pour la balise `option` correspondante (ce comportement est en fait défini carrément par HTML 4.01).

MSIE ne respecte aucun de ces deux comportements. C'est pourquoi il faut récupérer la valeur en passant par la collection `options` du `select`, sa propriété `selectedIndex`, et les propriétés `value` et `text` de l'option. Cet algorithme alourdi est présent dans notre fonction `getFieldValue`, non imprimée ici.

Une solution portable consiste à utiliser la fonction `$F` de Prototype pour récupérer la valeur d'un élément. Elle gérera même correctement les valeurs d'une liste à sélection multiple !

Les principaux points problématiques

Si on fait le bilan de ce que nous avons appris sur le DOM, on constate qu'on a à portée de main une puissance de traitement extraordinaire, avec la possibilité d'explorer les moindres recoins du document, et de le manipuler comme bon nous semble.

Hélas ! Le prix de ces possibilités est un peu lourd à payer :

- La mauvaise prise en charge du DOM niveau 2 (ou même niveau 1, pour certains points) par MSIE pose de graves problèmes de portabilité, nous forçant à alourdir considérablement notre code.
- D'autres navigateurs ont aussi quelques accrocs dans leur prise en charge du DOM, bien que sur des points beaucoup plus bénins.
- Les codes créant des fragments de DOM sont généralement verbeux : les interfaces qui nous sont fournies sont puissantes, mais il faut beaucoup de code pour créer quoi que ce soit, même des fragments très simples, du type : `<img src="logo.png" />` ou `<h1>Merci !</h1>`.
- Il est impossible de réaliser une extraction sur la base d'une règle CSS, alors que c'est la syntaxe de sélection que connaissent le mieux les développeurs web.

Toutefois, des solutions existent, même si elles ne sont pas forcément fournies par un standard W3C.

Ainsi, Prototype simplifie beaucoup de choses et ajoute de nombreuses possibilités. Nous allons nous régaler au chapitre suivant, en découvrant toute cette puissance conçue pour être facile d'emploi !

La bibliothèque script.aculo.us, basée sur Prototype, fournit un objet `Builder` conçu spécialement pour simplifier la création de fragments DOM. Nous ne l'étudierons pas dans le chapitre 7, qui couvre la majorité de script.aculo.us en détail, mais des exemples clairs sont disponibles sur le site de la bibliothèque.

Pour aller plus loin

Livres

DOM Scripting
Jeremy Keith
Friends of ED, septembre 2005, 341 pages
ISBN 1-590-59533-5

Sites

- DOM Scripting, le site :
  http://domscripting.com
- Le site de Jeremy Keith, gourou du scripting DOM :
  http://adactio.com
- Le site de Peter-Paul Koch, est un autre gourou du sujet :
  http://www.quirksmode.org
- La Mozilla Developer Connection a une référence complète de JavaScript et des informations sur le « DOM navigateur » (objets `navigator`, `window`, etc.) :
  - http://developer.mozilla.org/en/docs/JavaScript
  - http://developer.mozilla.org/fr/docs/JavaScript
  - http://developer.mozilla.org/fr/docs/DOM
- Les spécifications sont le point de référence incontournable !
  http://www.w3.org/DOM/DOMTR
- Le Web Standards Project (WaSP) fait avancer les standards qui comptent et sa DOM Scripting Force abat un boulot titanesque :
  http://webstandards.org/action/dstf

4

Prototype : simple, pratique, élégant, portable !

Sortie du cerveau fécond de Sam Stephenson (merci, merci, merci Sam !), Prototype est une bibliothèque JavaScript qui simplifie énormément la majorité des utilisations courantes de JavaScript, même lorsqu'il s'agit de fonctions avancées.Très bien structurée et organisée avec beaucoup de cohérence, Prototype accroît considérablement les capacités des classes incontournables (`Array`, `String`, `Number`) et des éléments HTML. Et tout ceci, en assurant une excellente portabilité d'un navigateur à l'autre, ce qui constitue d'ordinaire une épine dans le pied des développeurs web.

Ce qui est certain, c'est que les débutants JavaScript comme les gourous expérimentés prendront tout à coup beaucoup plus de plaisir à écrire du JavaScript s'ils utilisent Prototype. Alors vous aussi, faites-vous plaisir ; il vous suffit de lire ce chapitre.

Avant de commencer…

Ce chapitre va documenter Prototype d'un point de vue utilisateur. C'est-à-dire qu'on ne s'intéressera pas aux mécanismes internes de la bibliothèque, comme les méthodes `Object.extend`, `Class.create` ou `Element.extend`, pour ne citer qu'elles. On n'ira pas non plus fouiller dans les champs et méthodes privés des objets (généralement préfixés d'un tiret bas, par exemple `_observeAndCache` ou `_nativeExtensions`), qui appartiennent aux détails d'implémentation.

En effet, connaître ces rouages ne nous est d'aucune utilité pour utiliser correctement la bibliothèque. Ce chapitre vous dit déjà tout ce que vous avez besoin de savoir pour tirer parti au mieux des fonctionnalités proposées. Qui plus est, les détails d'implémentation peuvent changer considérablement d'une version à l'autre (ce sera d'ailleurs le cas pour la version 2.0). Seule l'interface de la bibliothèque reste à peu près stable.

Prototype propose aussi quelques classes tout à fait remarquables pour encapsuler les traitements Ajax, mais nous attendrons le chapitre 6 pour les aborder, après avoir décortiquer le fonctionnement de `XMLHttpRequest`. Nous pourrons alors répondre aux questions qu'aurait soulevées l'exposé de ces classes s'il avait été conduit dans ce chapitre.

Enfin, sachez que la plupart des exemples de plus de 3 lignes de ce chapitre sont fournis avec leurs pages de test dans l'archive des codes source pour ce livre, disponible sur le site des éditions Eyrolles.

Un mot sur les versions

À l'heure où j'écris ces lignes, la version publique de Prototype est la version 1.4.0, publiée en janvier 2006. Toutefois, la version 1.5.0 finale est prévue prochainement, et la version actuelle est la RC1 (*Release Candidate one*), marquée au 4 septembre 2006.

La version 1.5 a ajouté une très grande quantité de fonctions par rapport à la version 1.4.0, par exemple les fonctionnalités de sélection sur classe et de gestion des modèles, des améliorations à la couche Ajax, l'extension des éléments du DOM ou encore la correction d'un bogue de fuite mémoire important sur MSIE.

C'est pourquoi ce chapitre documente la version 1.5.0, sur la base du code source de la version 1.5.0 RC1. Vous êtes ainsi à jour sur les jolies nouveautés.

Vous pouvez vous procurer Prototype de plusieurs façons :

- Télécharger l'archive ou le fichier JavaScript sur le site officiel (c'est peut-être encore seulement la version 1.4.0) : http://prototype.conio.net/

- Récupérer la dernière version de la bibliothèque script.aculo.us, étudiée en détail au chapitre 7. Elle inclut Prototype dans sa dernière variante 1.5.0 connue (à ce jour, 1.5.0_rc1 avec quelques correctifs supplémentaires) :
  http://script.aculo.us/downloads
- L'archive des codes source pour ce livre contient dans chaque sous-répertoire de ce chapitre une version 1.5.0_rc1 avec quelques correctifs supplémentaires.

L'objet global Prototype

Commençons par l'objet global `Prototype`. Il s'agit d'ailleurs plus d'un espace de noms que d'un objet. On y trouve d'abord `Prototype.Version`, une constante indiquant la version exacte de Prototype, par exemple `1.5.0_rc1`. C'est utile pour vérifier une dépendance. La bibliothèque script.aculo.us, par exemple, s'en sert pour vérifier qu'on l'utilise avec une version suffisamment récente de Prototype. Ainsi, la version 1.6.4, qui dépend de Prototype 1.5.0, contient le fragment de code suivant :

Listing 4-1 Exemple de vérification de la version de Prototype

```
if ((typeof Prototype == 'undefined' ||
    ...
    parseFloat(Prototype.Version.split(".")[0] + "." +
               ➥ Prototype.Version.split(".")[1]) < 1.5)
    throw("script.aculo.us requires the Prototype ... >= 1.5.0");
```

Par ailleurs, `Prototoype` renferme deux fonctions simplistes, fort utiles dans de nombreux emplois de méthodes nécessitant un itérateur (voir la prochaine section) :

- `Prototype.emptyFunction` est une fonction vide, comme son nom l'indique. Elle ignore ses arguments, ne fait rien, et ne renvoie rien. Prototype s'en sert souvent pour éviter d'avoir à gérer le cas d'une fonction optionnelle qui n'aurait pas été fournie, en basculant sur celle-ci au moyen d'un simple opérateur `||`. Nous verrons un exemple de cela au chapitre 8, dans notre exemple autour de Flickr.
- `Prototype.K` est la fonction identité : elle renvoie simplement son premier argument. Elle est utilisée par de nombreuses méthodes à itérateur, lorsqu'un itérateur spécifique n'est pas fourni. Dans les prochaines sections, vous verrez de nombreuses méthodes ayant un itérateur optionnel en argument. S'il n'est pas fourni, c'est `Prototype.K` qui est utilisé à sa place.

Vocabulaire et concepts

Commençons par rappeler qu'en JavaScript, il n'y a pas à proprement parler de classes : tout est un objet. L'héritage est obtenu par ajout de champs et méthodes aux propriétés du prototype d'un objet. Nous avons couvert cela au chapitre 2.

Espaces de noms et modules

J'aurai souvent recours aux termes « espace de noms » et « module », que j'utilise ici de façon presque interchangeable. Lorsque je qualifie un objet d'espace de noms, je veux dire qu'il n'est pas destiné à être instancié, ni même incorporé au prototype d'un autre objet. Il n'existe que pour donner un contexte nommé à ses méthodes, et leur éviter ainsi d'être globales, mais aussi d'avoir à utiliser un nom plus long.

On trouve ainsi par exemple les objets `Ajax`, `Element`, `Abstract` et `Insertion`, qui ne constituent que des espaces de noms, comme un *namespace* C++, un paquet Java ou une unité Delphi.

Je qualifie par ailleurs certains objets de modules. Ces objets servent à regrouper des méthodes autour d'un aspect particulier, lesquelles sont vouées à être incorporées aux prototypes d'autres objets, afin de leur ajouter cet aspect. J'emploie ici le terme « aspect » au sens de la programmation orientée aspects (AOP). Le terme module prend ici le sens exact qu'il a, par exemple, en Ruby.

Prototype fournit plusieurs modules, dont `Enumerable` et `Element.Methods`.

Itérateurs

Vous allez aussi rencontrer le terme « itérateur » à tout bout de champ. Dans le cadre de Prototype, je qualifierai d'itérateur une fonction destinée à être invoquée sur chaque élément d'une itération. Cette itération est réalisée par une fonction qui reçoit la nôtre en argument. Ainsi, `Prototype.K` est exclusivement utilisée, en interne, comme itérateur par défaut. Il s'agit là d'une légère différence avec le sens classique du mot, qui désigne le mécanisme d'itération, et non celui d'opération sur les éléments produits par l'itération.

Les itérateurs permettent de séparer la logique d'itération de celle d'action, qui opère sur chaque élément. Vous les trouverez dans la plupart des bibliothèques standardisées de classes, mais il ne s'agit pas toujours de la fonction opérative, comme ici avec Prototype.

En C++, un itérateur est un objet encapsulant la logique d'itération, et non celle d'opération. Il suffit généralement qu'il fournisse les opérateurs `*`, `->` et `++`. En Java aussi, l'itérateur encapsule l'itération elle-même, sous forme de l'interface `java.util.Iterator`. En Ruby, un itérateur est une méthode encapsulant l'itération,

qui reçoit un bloc ou une `Proc` en argument, auquel la méthode passe tour à tour chaque élément de l'itération. Vous trouverez des équivalents en Perl, Python, C#, etc.

Élément étendu

Prototype estime que les éléments HTML, tels que les fournit le DOM, sont un peu nus. Il est vrai qu'au regard des utilisations communes qu'on en fait, l'interface `HTMLElement` du DOM niveau 2 HTML (voir chapitre 3) est assez légère. Prototype fait donc tout son possible pour enrichir les éléments qu'il vous renvoie.

Si le navigateur offre un mécanisme de prototype pour les éléments du DOM, Prototype enrichit automatiquement tous les éléments des méthodes présentes dans le module `Element.Methods`, que nous verrons en détail plus loin.

Par ailleurs, tout élément de type champ de formulaire reçoit également les méthodes de `Form.Element.Methods` (toutes celles de `Form.Element` moins `focus` et `select`, Dieu sait pourquoi...). Dans le même esprit, tout élément `form` reçoit également les méthodes de `Form.Methods` (toutes celles de `Form` sauf `reset`, même remarque).

Si cette possibilité lui est refusée (par exemple sur MSIE), il enrichira à la volée tout élément accédé au travers de ses incontournables fonctions `$`, `$$`, et `getElementsByClassName` (tant dans `document` que dans `Element`), ce qui en pratique signifie que dans la vaste majorité des cas, ces méthodes sont en accès direct sur les éléments manipulés. Sur un tel élément `elt`, au lieu de faire :

```
Element.methodeSympa(elt)
Element.autreMethodeSympa(elt, arg1, arg2…)
```

Vous pouvez directement faire :

```
elt.methodeSympa()
elt.autreMethodeSympa(arg1, arg2…)
```

Ce qui est plus sympathique, plus orienté objet, et plus court !

Nous y reviendrons dans la section consacrée à `Element.Methods`. D'ici là, gardez simplement ceci à l'esprit :

- Partout où j'indique qu'une fonction prend en argument un élément (ou plusieurs), vous pouvez passer soit un élément existant, soit son ID : de tels arguments sont toujours utilisés au travers de la fonction `$`, que nous allons voir dans un instant. Ces éléments sont donc étendus quoi qu'il arrive.
- Quand j'écris qu'une fonction renvoie un « élément étendu » (ou un tableau de tels éléments), ces éléments sont dotés directement des méthodes du module `Element.Methods`.

Alias

Dans un souci de confort maximal, Prototype donne parfois des alias à certaines méthodes, lorsque celles-ci existent fréquemment sous deux noms différents dans les bibliothèques les plus répandues. Ainsi, `collect` et `map` sont synonymes, de même que `detect` et `find`, `select` et `findAll`, ou encore `include` et `member`. L'objectif est de vous permettre d'utiliser une méthode par un nom qui vous est familier, pour faciliter votre apprentissage.

Dans de tels cas :

- Je précise les deux noms dans les titres de section.
- J'indique dans le corps du texte que les méthodes sont des alias.
- Je précise l'alias dans les blocs de syntaxe à l'aide d'un slash (/), comme ceci :

```
objet.nom1/nom2(arguments)
```

Comment utiliser Prototype ?

C'est très simple. Prototype est fourni sous la forme d'un unique fichier de script, `prototype.js`, d'environ 61 Ko (moins d'une seconde de téléchargement, et le cache de votre navigateur le gardera bien au chaud par la suite). Il vous suffit de charger ce script depuis l'en-tête de votre page (X)HTML, de préférence avant les autres scripts, ces derniers ayant tout intérêt à s'en servir :

```
<head>
    …
    <script type="text/javascript" src="…/prototype.js"></script>
    …
</head>
```

C'est tout !

Enfin, sachez que l'archive des codes source pour ce livre, disponible sur le site des éditions Eyrolles, contient de nombreuses pages complètes d'exemples pour l'ensemble des fonctionnalités présentées dans ce chapitre. Je vous invite à la télécharger et à la décompresser, si ce n'est déjà fait, pour pouvoir découvrir chaque objet en action, en plus des exemples succincts contenus dans le texte.

Vous allez aimer les dollars

Loin de moi l'idée de vous taxer de mercantilisme, mais il se trouve que Prototype a choisi, pour des raisons de confort, de fournir ses fonctions les plus utiles (que vous trouverez vite indispensables, voire incontournables !) sous des noms les plus courts possibles. Le meilleur moyen pour éviter les conflits avec des noms existants était l'emploi du préfixe `$`, dont on n'oublie trop souvent qu'il est autorisé comme caractère d'identifiant en JavaScript (ECMA-262, 7.6§2).

Il existe donc six fonctions globales d'intérêt majeur (cinq en Prototype 1.4.0) : `$` (oui, juste `$` ! Il devient difficile de faire plus court, et donc plus discret, dans votre code !), `$A`, `$H`, `$F`, `$R` et `$$`. Pour les décrire, je vais parfois devoir faire appel à des concepts qui seront décrits plus loin dans ce chapitre, mais cela ne devrait pas poser de problèmes de compréhension.

La fonction $ facilite l'accès aux éléments

C'est sans conteste la fonction la plus utilisée de Prototype. Son objectif : vous permettre d'accéder aux éléments du DOM par leur ID, et réduire au maximum les cas particuliers et les tests. Je vais m'étendre un peu sur sa description, car elle constitue vraiment une pierre angulaire de Prototype, et on l'utilise souvent en-dessous de ses capacités.

Voici ses invocations possibles :

```
$(id) -> objÉtendu
$(obj) -> objÉtendu
$(id, id, obj…) -> [objÉtendu, objÉtendu, objÉtendu…]
```

À l'avenir, je résumerai ce genre de possibilités comme ceci :

SYNTAXE

```
$((id/obj)…) -> objÉtendu / [objÉtendu…]
```

Si vous lui passez un ID (une `String`), elle récupère l'élément avec un `document.getElementById`. Sinon, elle considère que vous lui passez en réalité l'élément lui-même. Dans tous les cas, elle enrichit l'élément si besoin pour s'assurer que vous récupérez au final un élément étendu.

La plupart des fonctions de Prototype qui acceptent des éléments en arguments commencent par les passer à `$`, histoire d'accepter tant des ID que des éléments déjà récupérés, et d'être certaines d'utiliser leur version étendue. Je ne préciserai donc plus, à l'avenir, qu'une fonction prend un ID ou un élément : je dirai juste « un élément ».

Si vous ne lui avez passé qu'un élément, elle vous renvoie l'élément étendu. Si elle a reçu plusieurs arguments, elle renvoie un tableau des éléments étendus correspondants. Si vous souhaitez constituer un tableau d'éléments, il est donc inutile de faire :

```
// Code inutilement complexe !
var items = []
items[0] = $('item0');
items[1] = $('item1');
items[2] = $('item2');
```

Préférez :

```
// Code bien plus pratique
var items = $('item0', 'item1', 'item2');
```

La fonction $A joue sur plusieurs tableaux

Nous allons le voir dans quelques pages, l'objet natif JavaScript `Array` est très développé par Prototype, qui y incorpore notamment le module `Enumerable`. Un tableau est donc bien plus puissant que la plupart des collections renvoyées par le DOM (notamment les objets `NodeList` et `HTMLCollection`), qui n'offrent pour la plupart que les propriétés `length` et `item`.

C'est pourquoi la fonction `$A` est importante, et souvent utilisée. Elle prend un argument susceptible d'être transformé en tableau, et en fait un objet `Array` digne de ce nom, doté de toutes les extensions dues à `Enumerable` !

SYNTAXE

```
$A(obj) -> tableau
```

Reste à savoir ce qui constitue un argument susceptible d'être transformé en tableau. `$A` fonctionne comme ceci :

1. Si on lui passe `null` ou `undefined`, elle renvoie un tableau vide.
2. Si l'objet reçu implémente la méthode `toArray()`, elle utilise cette méthode. Cela permet à nos classes de contrôler leur transformation en tableau, si nécessaire. C'est notamment le cas de tout objet incorporant le module `Enumerable`.
3. Sinon, l'objet passé doit disposer d'une propriété `length` et de propriétés d'indice numérique, afin de pouvoir être indexé comme un tableau. Chaque propriété à laquelle on accédera ainsi sera ajoutée au tableau résultat.

Voici un exemple complet de tous les cas d'utilisation :

Listing 4-2 $A dans tous ses états !

```
$A(null)
// => []
$A()
// => []
$A([])
// => []
$A(['et', 'hop', 'facile !'])
// => ['et', 'hop', 'facile !']

var buddy = {
    firstName: 'Amir',
    lastName: 'Jaballah',

    toArray: function() {
        return [this.firstName, this.lastName];
    }
};

$A(buddy)
// => ['Amir', 'Jaballah']

var convoluted = {
    0: 'eh bien',
    1: 'voilà un exemple',
    2: 'pour le moins tordu !',
    length: 3
};

$A(convoluted)
// => ['eh bien', 'voilà un exemple', 'pour le moins tordu !']
```

La fonction $H, pour créer un Hash

Nous verrons plus loin que Prototype définit un objet sympathique, `Hash`, qui représente un tableau associatif (un peu comme `java.util.HashMap` ou les tableaux de PHP, par exemple). Si vous avez lu attentivement le chapitre 2, vous savez qu'un objet JavaScript est, essentiellement, un tableau associatif dont les clés sont les noms des propriétés, tandis que les valeurs sont… eh bien, les valeurs des propriétés. Au premier abord, `Hash` n'y ajoute donc rien.

Nous verrons en réalité que si. Et c'est un type de données si fréquemment utilisé qu'il dispose de sa propre fonction de conversion, `$H`. En réalité, l'un ne va pas sans l'autre : il n'existe pas d'autre moyen que cette fonction pour obtenir une instance de `Hash`.

SYNTAXE

```
$H([objet]) -> Hash
```

On peut donc créer un `Hash` vierge simplement avec `$H()`, ou obtenir la version `Hash` d'un objet existant pour l'examiner plus confortablement, en faisant `$H(objet)`. La fonction s'assure par ailleurs que l'objet résultat incorpore le module `Enumerable`, ce qui ajoute encore aux possibilités.

La fonction $F, des valeurs qui sont les vôtres

Si vous avez déjà essayé de récupérer de façon générique la valeur d'un champ, vous savez que ce n'est pas si trivial. La plupart des types de champs fournissent une propriété `value`, mais on ne doit pas la prendre en compte pour des cases à cocher et boutons radio décochés, et `select` soulève plusieurs questions, suivant qu'il autorise une sélection multiple ou non, ou qu'on utilise MSIE et que la propriété `value` de ses options est mal implémentée. Dans le dernier exemple du chapitre précédent, le code source de la fonction `getFieldValue` illustre bien cette complexité.

Avec Prototype, il suffit d'écrire `$F(élément)`. C'est tout. Je le reprends ci-dessous comme code individuel, pour vous faciliter la lecture en diagonale à l'avenir :

SYNTAXE

```
$F(element) -> valeurEnTexte / [valeurEnTexte…]
```

Petite précision toutefois : si l'élément indiqué est un `select` à sélection multiple, on récupère un tableau des valeurs pour les options sélectionnées.

La fonction $R et les intervalles

Un des objets sous-utilisés de Prototype est `ObjectRange`, qui permet de représenter un intervalle de valeurs pour n'importe quel objet, du moment que celui-ci fournit une méthode `succ()`. La manière la plus simple de créer un `ObjectRange` est d'utiliser la fonction `$R`. C'est très utile pour séparer une définition de boucle de son utilisation.

Nous reviendrons sur `$R` dans la documentation d'`ObjectRange`, mais voici tout de même sa syntaxe :

SYNTAXE

```
$R(debut, fin[, finExclue = false]) -> ObjectRange
```

L'objet ObjectRange renvoyé représente une itération entre debut et fin, laquelle peut être exclue si vous précisez un troisième argument true. Exemple :

```
var loop = $R(1, 42);
...
loop.each(function(i) { // appelée pour i de 1 à 42 } )
```

La fonction $$ et les règles CSS

Apparue avec Prototype 1.5.0, $$ permet de récupérer tous les éléments correspondant à des règles CSS. Il s'agit en fait une fonction d'enrobage autour du nouvel objet Selector, qui représente, comme son nom l'indique, un sélecteur CSS. La syntaxe est la suivante :

SYNTAXE

```
$$(règle…) -> [objÉtendu…]
```

On récupère un tableau de tous les éléments correspondants aux diverses règles. Attention toutefois à l'emploi de plusieurs règles : si des éléments correspondent à plusieurs de ces règles, ils apparaîtront plusieurs fois dans le tableau résultat. Les éléments retournés sont garantis étendus, comme pour $.

Si vous souhaitez, pour des raisons de performances, conserver une règle précompilée dans un Selector pour l'utiliser de multiples fois, sachez que le constructeur de Selector ne gère pas le sélecteur d'éléments descendants (l'espace) : c'est $$, via Selector.findChildElements, qui utilise une astuce pour implémenter ce sélecteur à moindre coût. Nous reviendrons dans la section dédiée à Selector sur les syntaxes prises en charge directement par celui-ci.

Voici un exemple de code qui masque tous les paragraphes ayant une classe toggle dans un conteneur d'ID demo :

```
$$('#demo p.toggle').each(function(p) {
    p.hide(); // hide() disponible car p est un élément étendu
});
```

Jouer sur les itérations avec $break et $continue

Nous allons aborder tout au long de ce chapitre de nombreuses méthodes encapsulant des itérations, par exemple each, collect, inject ou map. Ces méthodes ont recours à des itérateurs pour traiter tour à tour les objets sur lesquels on boucle.

SYNTAXE

```
throw $break;
throw $continue;
```

Dans une boucle classique, le langage fournit deux moyens de court-circuit, présents dans de nombreux autres langages et signalés au chapitre 2 : `break` et `continue`. Le premier « casse » la boucle, l'exécution continuant à la première instruction après la boucle. Le second court-circuite juste le tour courant, faisant immédiatement passer la boucle à l'itération suivante.

Lorsqu'on utilise des itérateurs, ces mots réservés de JavaScript ne nous sont plus d'aucune utilité, puisque le code réalisant la boucle et celui réalisant le traitement sont dans deux fonctions distinctes : la méthode d'itération et l'itérateur, respectivement.

Afin de vous fournir néanmoins ces possibilités de court-circuit, Prototype définit donc deux « exceptions globales » : `$break` et `$continue`. Pour obtenir un résultat équivalent aux mots réservés, il vous suffit dans votre itérateur de lancer l'exception correspondante.

Voici par exemple un code qui récupère dans une liste les 5 premiers candidats ayant plus de 15 de moyenne, et court-circuite l'itération dès ce nombre est atteint.

Listing 4-3 Court-circuit dans un itérateur avec $break

```
// Chaque candidat a des propriétés name et grade, et candidates
// est un gros tableau de candidats.
function get5FirstGoodCandidates(candidates) {
    var count = 0;
    var result = [];
    candidates.each(function (c) {
        if (c.grade >= 15) {
            result.push(c);
            if (5 == ++count)
                throw $break;
        }
    });
    return result;
} // get5FirstGoodCandidates
```

Extensions aux objets existants

J'ai classé les apports de Prototype en deux grandes catégories : d'abord les extensions aux objets existants, ensuite les ajouts purs et simples. Nous allons voir les extensions d'abord, parce qu'il s'agit sans doute là du plus grand impact sur votre code. En effet, il s'agit d'objets dont vous croyez connaître toutes les possibilités, car ils vous sont familiers. Le risque de sous-exploiter leurs nouvelles capacités est donc élevé. Et ce serait vraiment dommage...

Un Object introspectif

`Object` est un espace de noms qui fournit quelques méthodes sympathiques.

SYNTAXE

```
Object.clone(objet) -> cloneDeObjet
Object.inspect(objet) -> représentationString
Object.keys(objet) -> Enumerable
Object.values(objet) -> Enumerable
```

`Object` est doté d'une petite méthode nommée `inspect`, destinée aux développeurs web en train de mettre leur code au point. Elle tente de fournir une représentation textuelle la plus efficace possible ; en effet, la représentation par défaut des objets sous forme de `String` laisse souvent à désirer.

Ainsi, les tableaux apparaissent soit sous forme de leurs valeurs séparées par des virgules (donc un affichage totalement vide pour un tableau vide, ou incompréhensible pour `[' ', '', '      ', '\t\n']`), soit sous la très inutile forme « `[Array]` »... `String` ne fait guère mieux. `undefined` et `null` sont le plus souvent invisibles, et les objets, n'en parlons pas.

Voici comment procède `Object.inspect` :

- `null` et `undefined` donnent `'null'` et `'undefined'`. Enfin, en théorie, parce qu'à l'heure actuelle, l'utilisation malheureuse de == au lieu de === fait que les deux donnent `'undefined'`...
- Si l'objet dispose d'une méthode `inspect()`, celle-ci est appelée (Prototype en fournit pour `String`, `Enumerable`, `Array`, `Hash` et `Element`, ce qui couvre l'essentiel des besoins).
- À défaut, la méthode `toString()` est appelée. Outre celles spécifiques aux objets `Element.ClassNames` et `Selector`, il faut savoir que tout objet JavaScript a par défaut une méthode `toString()`, et ce depuis toujours (JavaScript 1.0).

Comparez quelques exemples de représentation standard avec `toString()` et de résultat avec `inspect()`. Vous allez sentir la différence en termes de débogage :

Listing 4-4 toString() versus inspect()

```
var tableau = [ '', 'Salut', 'C\'est génial !', '\n', '"TDD"' ];
var obj = { nom: 'Christophe', age: 28 };
var objH = $H(obj);

alert(tableau + '\n' + obj + '\n' + objH);
alert(Object.inspect(tableau) + '\n' +
Object.inspect(obj) + '\n' +
Object.inspect(objH));
```

Voici les résultats :

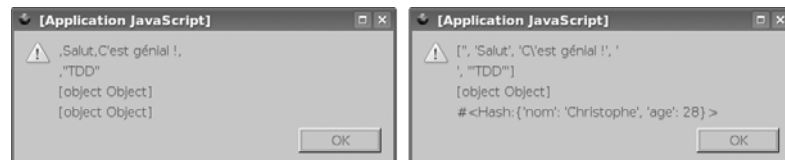


Figure 4–1
toString() versus inspect()

Il y a déjà du mieux... En combinant avec l'objet global `console` fourni par Firebug (voir chapitre 2), on a de quoi faire.

Les méthodes `keys()` et `values()` permettent de traiter un objet comme un hash d'associations clé/valeur : elles renvoient un tableau des noms de propriétés, ou de leurs valeurs, respectivement :

```
Object.keys(tableau) // => [0, 1, 2, 3]
Object.keys(obj) // => [ 'nom', 'age' ]
Object.values(obj) // => [ 'Christophe', 28 ]
```

Enfin, la méthode `clone` fournit, comme son nom l'indique, un clone exact de l'objet source.

Gérer correctement le binding

On a vu en détail, au chapitre 2, la question épineuse du *binding*. Il s'agit principalement de pouvoir passer une méthode en argument, pour exécution ultérieure, sans que celle-ci perde le lien qui l'attache à un objet particulier.

Prototype fournit deux ajouts à l'objet `Function` qui s'occupent spécifiquement du binding : `bind` et `bindAsEventListener`.

SYNTAXE

```
monObjet.methode.bind(monObjet[, arg…])
monObjet.methode.bindAsEventListener(monObjet[, arg…])
```

La méthode `bind` permet de « transformer » une méthode classique en méthode à binding garanti, c'est-à-dire sur laquelle `this` vaudra toujours ce que vous aurez précisé au moment du `bind`. Qui plus est, elle peut lui fournir des arguments préremplis, auxquels les arguments d'invocation seront ajoutés (pas de remplacement !).

Voyez l'exemple suivant.

Listing 4-5 Avec ou sans bind…

```
var obj = {
    location: 'Paris, France',

    getLocation: function() {
        return this.location;
    } // getLocation
};

function show(getter) {
    alert(getter());
} // show

show(obj.getLocation);
// => this global (window) : window.location => l'URL de la page
show(obj.getLocation.bind(obj));
// => this sera bien obj : obj.location => « Paris, France »
```

Ce qu'il faut bien comprendre, c'est que `bind` renvoie une nouvelle fonction, qui peut être appelée en lieu et place de l'ancienne autant de fois qu'on veut.

On trouve un exemple de `bind` avec un argument prédéfini dans l'étude de cas sur l'API REST Flickr au chapitre 8, pour l'objet de chargement parallèle `gXSLT`.

Il existe un cas particulier : celui des gestionnaires d'événements. Lorsque vous utilisez des fonctions globales comme gestionnaire, vous n'avez rien de spécial à faire. Toutefois, quand vous passez une méthode et que celle-ci a besoin de référencer son instance conteneur avec `this`, il est important de fournir un enrobage approprié de la méthode à l'aide de `bindAsEventListener`. En effet, `bind` ne suffira pas sur de vieux navigateurs, car ils ne fourniraient pas toujours à votre gestionnaire l'événement courant comme premier argument.

Là aussi, un exemple met les choses au clair :

Listing 4-6 Avec ou sans bindAsEventListener...

```
var obj = {
    location: 'Paris, France',

    handleLeftClick: function(event) {
        if (!event) {
            alert("Ah, pas d'objet événement...");
            return;
        }
        Event.stop(event);
        if (Event.isLeftClick(event))
            alert('Location : ' + this.location);
    } // handleLeftClick
};

Event.observe('btnTestBasic', 'click', obj.handleLeftClick);
// => this = bouton : btnTestBasic.location => undefined
Event.observe('btnTestBind', 'click',
    obj.handleLeftClick.bind(obj));
// => this correct, mais sur de vieux navigateurs, argument
//    event manquant...
Event.observe('btnTestBAEL', 'click',
    obj.handleLeftClick.bindAsEventListener(obj));
// => this correct, event garanti :-)
```

Des drôles de numéros

Le vénérable objet `Number`, utilisé par tous les nombres, y compris les littéraux, dispose de quelques menues extensions.

SYNTAXE

```
variableNombre.toColorPart() -> HexaDeuxCaractères
variableNombre.succ() -> nombreSuivant
variableNombre.times(function(index) { … })
(literalEntier).toColorPart()
(literalEntier).succ()
(literalEntier).times(function(index) {…});
```

Tout d'abord la méthode `toColorPart()`, qui renvoie la représentation hexadécimale du nombre. Elle est surtout censée être utilisée sur des nombres de 0 à 255, et destinée à composer la représentation CSS d'une couleur.

Voici un exemple typique avec `Array.inject`, que nous étudierons plus loin :

```
var rgb = [ 128, 255, 0 ];
var cssColor = rgb.inject('#', function(s, comp) {
    return s + comp.toColorPart();
});
// cssColor == '#80ff00'
```

Les méthodes `succ()` et `times(iterator)` permettent aux nombres de réaliser des itérations dans le plus pur style Ruby. En effet, la méthode `times` synthétise une itération de 0 jusqu'au nombre (exclu, ou borne ouverte si vous avez l'esprit mathématique). Elle utilise pour ce faire un `ObjectRange`, qui a besoin que l'objet d'origine implémente une méthode `succ`, laquelle fournit la valeur suivante à chaque tour (ici, `succ()` renvoie la valeur appelante plus un).

Ainsi, pour réaliser une boucle de zéro à n exclu, au lieu de faire :

```
for (var index = 0; index < n; ++index)
    // code
```

On peut faire :

```
n.times(function(index)) {
    // code
}
```

Ce qui est plutôt lisible, vous ne trouvez pas ? « n times »...

Attention toutefois : en raison de la syntaxe des nombres, qui utilisent déjà le point (`.`) comme séparateur décimal, on ne peut pas invoquer une méthode directement sur un littéral entier : `5.succ()` génère une erreur de syntaxe. On doit donc, comme mentionné plus haut dans le bloc de syntaxe, recourir à une astuce pour lever l'ambiguïté au niveau de l'analyseur. La plus simple est d'encadrer le nombre entre parenthèses. Il en existe deux autres, que je vous laisse chercher si vous êtes du genre curieux...

Un objet String très enrichi

`String` est l'un des objets les plus enrichis par Prototype. L'autre est `Array`, que nous examinerons tout à l'heure. Il faut avouer que malgré la relative richesse de ses méthodes originelles, `String` a pas mal de lacunes au regard de son usage courant. En raison du grand nombre d'extensions, j'ai découpé l'examen des nouvelles méthodes par thème.

Toutes les méthodes de modification renvoient en réalité une nouvelle version modifiée : elles ne touchent pas à la chaîne de caractères originale.

Retraits de caractères : strip, stripTags, stripScripts, truncate

Ces quatre petites méthodes renvoient une version purgée ou tronquée.

SYNTAXE

```
chaine.strip() -> chaine
chaine.stripTags() -> chaine
chaine.stripScripts() -> chaine
chaine.truncate([longueur = 30[, troncature = '...']]) -> chaine
```

La méthode `strip()` équivaut au plus classique `trim()` : elle retire les espacements (*whitespace*) en début et en fin de texte.

La méthode `stripTags()` purge le texte de toute balise ouvrante et fermante, attributs compris. Ne pas confondre avec `escapeHTML`.

La méthode `stripScripts()` se contente de supprimer les balises `script` et leur contenu, ce qui est très utile en termes de sécurité. Prototype s'en sert aussi beaucoup en interne, pour insérer un contenu HTML : il retire systématiquement les élément scripts pour les évaluer à part, immédiatement après insertion.

La méthode `truncate` est un tout petit peu plus complexe, car elle dispose de deux arguments optionnels. Il s'agit ici de tronquer un texte à une longueur maximale, en remplaçant la partie tronquée par un texte de substitution. On imagine facilement l'intérêt dans des affichages à largeur fixe. L'argument `longueur`, qui vaut 30 par défaut, donne la longueur maximale du texte résultat (texte de troncature compris), et le second argument indique le texte de troncature, qui vaut par défaut `'...'` (trois caractères point).

Voici quelques exemples d'utilisation.

Listing 4-7 Méthodes de retrait de caractères

```
var spacedOutText = '   Bonjour monde\n';
var markup = '<h1>Texte balisé</h1>\n' +
    '<p>Vous voyez, il y a des <strong>balises</strong>.</p>';
var scriptMarkup = '<h1>Texte balisé</h1>\n' +
    '<p>Vous voyez, il y a des <strong>balises</strong>.</p>\n' +
    '<script type="text/javascript">\n' +
    'window.location.href = "http://site-de-pirate.com/
pique_mon_ip";\n' +
    '</script>\n' +
    '<p>Fin du balisage</p>';
var longText = 'Ceci est un texte un peu trop long pour moi';
```

```
spacedOutText.strip()
// 'Bonjour monde'
markup.stripTags()
// 'Texte balisé\nVous voyez, il y a des balises.'
scriptMarkup.stripTags()
// 'Texte balisé\nVous voyez, il y a des balises.\n\n' +
// 'window.location.href = ...\n\n' +
// 'Fin du balisage'
scriptMarkup.stripScripts()
// '<h1>Texte balisé</h1>\n' +
// '<p>Vous voyez, il y a des <strong>balises</strong>.</p>\n' +
// '\n<p>Fin du balisage</p>'
longText.truncate()
// => 'Ceci est un texte un peu tr...'
longText.truncate(42)
// => 'Ceci est un texte un peu trop long pour...'
longText.truncate(42, '…')
// => 'Ceci est un texte un peu trop long pour m…'
```

Transformations : sub, gsub, escapeHTML, unescapeHTML, camelize

Ces méthodes permettent d'effectuer des remplacements à comportement variable ou fixe sur le texte.

SYNTAXE

```
chaine.camelize() -> chaine
chaine.escapeHTML() -> chaine
chaine.gsub(pattern, replacement|iterator) -> chaine
chaine.sub(pattern, replacement|iterator[, count = 1]) -> chaine
chaine.unescapeHTML() -> chaine
```

Commençons par les plus simples. La méthode `escapeHTML()` « désamorce » un code HTML : le HTML est utilisé littéralement au lieu d'être interprété par le navigateur. La méthode `unescapeHTML()` fait exactement l'inverse.

La méthode `camelize()` transforme un texte minuscule composé de parties séparées par des tirets (-) en texte *CamelCase* minuscule. Il ne s'agit pas ici d'une transformation spécifique aux textes à destination des utilisateurs : on cible spécifiquement le passage d'un nom de propriété CSS à son équivalent dans le DOM niveau 2 Style. D'ailleurs, cette méthode est utilisée en interne par les méthodes `getStyle` et `setStyle` de `Element.Methods`, que nous étudierons plus loin dans ce chapitre.

Les méthodes les plus avancées sont `sub` et `gsub`. En surface, elles semblent globalement équivalentes à la méthode existante `replace`. Elles sont en réalité un peu plus puissantes.

Prenons d'abord `gsub`. Elle va remplacer toutes les occurrences du motif `pattern` trouvées dans la chaîne. Ce motif est une expression rationnelle fournie de préférence directement comme objet `RegExp`, par exemple `/\w+/`. Ce qui peut varier, c'est le remplacement. On peut passer un texte ou un itérateur. Dans le texte, il est possible de référencer les groupes isolés par le motif à l'aide d'une syntaxe inspirée de Ruby : `#{numéro}`. L'itérateur, en revanche, sera appelé pour chaque correspondance, avec l'objet la représentant en argument. Il doit renvoyer le texte de remplacement.

Beaucoup de gens ignorent que la fonction native `String.match` renvoie non pas une chaîne de caractères, mais un objet `MatchData`, qui décrit la correspondance. Cet objet est une sorte de tableau, dont l'index 0 renvoie la correspondance totale, et dont les indices supplémentaires renvoient les groupes isolés par les couples de parenthèses du motif.

Voici un exemple :

```
var md = 'salut les grenouilles'.match(/^(.)(\w+)/);
md[0] // => 'salut'
md[1] // => 's'
md[2] // => 'alut'
```

Enfin, un mot sur la méthode `sub` : elle est similaire à `gsub`, à ceci près qu'elle peut ne remplacer qu'une partie des occurrences (par défaut, seulement la première), à l'aide de son argument optionnel `count`.

Voyons quelques exemples de nos nouvelles méthodes.

Listing 4-8 Transformations de texte

```
'border'.camelize()
// => 'border'
'border-style'.camelize()
// => 'borderStyle'
'border-left-color'.camelize()
// => 'borderLeftColor'
'<h1>Un joli titre</h1>'.escapeHTML()
// => '&lt;h1&gt;Un joli titre&lt;/h1&gt;'
'&lt;h1&gt;Un joli titre&lt;/h1&gt;'.unescapeHTML()
// => '<h1>Un joli titre</h1>'
'je m\'appelle charles-edouard'.gsub(/[aeiouy]/, '-')
// => 'j- m\'-pp-ll- ch-rl-s--d---rd'
'je m\'appelle charles-edouard'.gsub(/[aeiouy]/, '[#{0}]')
// => 'j[e] m\'[a]pp[e]ll[e] ch[a]rl[e]s-[e]d[o][u][a]rd'
'je m\'appelle charles-edouard'.gsub(/[aeiouy]/, '\##{0}')
// => 'j#e m\'#app#ell#e ch#arl#es-#ed#o#u#ard'
'charles-edouard de la prutenatilde'.gsub(/\w+/,
```

```
function(match) {
    return match[0].charAt(0).toUpperCase() +
           match[0].substring(1).toLowerCase(
})
// => 'Charles-Edouard De La Prutenatilde'
'charles-edouard de la prutenatilde'.sub(/\w+/, '[#{0}]')
// => '[charles]-edouard de la prutenatilde'
'charles-edouard de la prutenatilde'.sub(/\w+/, '[#{0}]', 2)
// => '[charles]-[edouard] de la prutenatilde'
```

Fragments de scripts : extractScripts, evalScripts

La méthode `extractScripts` est presque exactement l'inverse de `stripScripts` : elle renvoie un tableau des portions de script du texte (sans leurs balises `script`). La méthode `evalScripts` pousse cette logique un cran plus loin : elle extrait les scripts et les exécute à tour de rôle avec `eval`.

SYNTAXE

```
chaine.extractScripts() -> [texteScript…]
chaine.evalScripts() -> [résultatScript…]
```

Il est assez probable que vous n'utiliserez jamais ces méthodes directement. Elles sont en revanche très utilisées, en interne, par les mécanismes d'insertion (espace de noms `Insertion`, décrit vers la fin du chapitre) et sur les contenus (X)HTML récupérés par les requêtes Ajax.

Conversions et extractions : scan, toQueryParams, parseQuery, toArray, inspect

Commençons par les méthodes simples. Nous avons déjà évoqué `toArray()` en étudiant la fonction `$A`. Une `String` peut ainsi être traitée comme un tableau de caractères.

SYNTAXE

```
chaine.inspect([useDoubleQuotes = false]) -> chaine
chaine.parseQuery/toQueryParams() -> objet
chaine.scan(pattern, iterator) -> laMemeChaine
chaine.toArray() -> [caractère…]
```

La méthode `inspect()` a aussi été vue lorsque j'ai présenté `Object.inspect`. Ici, elle « échappe » simplement les *backslashes* et apostrophes, encadrant le tout entre guillemets simples ou doubles (je trouve d'ailleurs que ce n'est pas suffisant : à mon goût, elle devrait aussi échapper les retours chariot, sauts de ligne et tabulations).

La méthode `parseQuery` analyse une *query string*, cette portion d'une URL qui démarre avec le point d'interrogation (`?`) et s'arrête éventuellement avec l'ancre (`#`). C'est là que sont situés les paramètres transmis en `GET`. La méthode `parseQuery` extrait cette portion d'une URL et la découpe pour renvoyer un objet avec une propriété par paramètre, et bien sûr les valeurs correctement définies pour chaque propriété.

La méthode `toQueryParams` est en réalité un alias de `parseQuery` : les deux noms référencent la même méthode.

Enfin, la méthode `scan` est une variante restreinte de `gsub`, qui ne sert qu'à transmettre chaque correspondance d'un motif à un itérateur. Elle renvoie la chaîne d'origine au lieu d'un tableau de résultats.

Quelques petits exemples ? Allez, d'accord.

Listing 4-9 Conversions et extractions

```
var query = '?firstName=Christophe&age=28'.parseQuery();
alert(Object.inspect($H(query)));
// => '<#Hash:{'firstName': 'Christophe', 'age': 28}>'
alert(Object.inspect('salut'.toArray()));
// => ['s', 'a', 'l', 'u', 't']
alert(Object.inspect("Salut les p'tits loups \\o/ !"));
// => 'Salut les p\'tits loups \\o/ !'
var oCounts = [];
'foo boo boz'.scan(/o+/, function(match) {
    oCounts.push(match[0].length);
});
alert(Object.inspect(oCounts));
// => [2, 2, 1]
```

Des tableaux surpuissants !

Avant toute chose, il faut dire que `Array` incorpore le module `Enumerable`, que nous allons voir dans la section dédiée aux nouveautés apportées par Prototype. De base, un tableau dispose donc de toutes les possibilités fournies par ce module, ce qui fait déjà beaucoup.

Cette section se penche sur les méthodes qui sont ajoutées en plus de celles du module `Enumerable`. Et comme il y en a beaucoup, je les ai donc classées par thème.

Conversions : from, inspect

La méthode `Array.from` est un alias de la fonction `$A`. Vous pouvez utiliser alternativement l'une ou l'autre syntaxe, suivant votre esthétique personnelle.

La méthode `inspect()` a déjà été présentée en étudiant `Object.inspect`. Elle renvoie ici une représentation entre crochets, les éléments étant séparés par une virgule et une espace, chaque élément étant représenté par un `Object.inspect` sur sa valeur. On obtient donc un texte très proche de l'équivalent littéral JavaScript.

SYNTAXE

```
Array.from(obj) -> tableau
tableau.inspect() -> chaine
```

Quelques exemples :

```
Array.from('hello')
// => ['h', 'e', 'l', 'l', 'o']
Array.from({0: 'c\'est', 1: 'vraiment', 2: 'bizarre', length: 3})
// => ['c\'est', 'vraiment', 'bizarre']
[ 42, "hello", Number.POSITIVE_INFINITY, [] ].inspect()
// => [42, 'hello', Infinity, []]'
```

Extractions : first, last, indexOf

Quelques petites méthodes toutes simples ont été ajoutées pour unifier l'accès aux éléments.

SYNTAXE

```
tableau.first() -> objet
tableau.last() -> objet
tableau.indexOf(obj) -> nombre
```

La méthode `first()` renvoie le premier élément, ou `undefined` si le tableau est vide. La méthode `last()` raccourcit enfin la syntaxe pénible `tab[tab.length - 1]`, et renvoie donc le dernier élément (ou `undefined` si le tableau est vide).

La méthode `indexOf`, définie par ailleurs dans le standard JavaScript, mais souvent manquante dans la pratique, raccourcit l'éternelle boucle de recherche en renvoyant la position de l'argument dans le tableau. La comparaison se fait avec l'opérateur `==`, et la fonction renvoie `-1` si la valeur n'a pas été trouvée.

```
[].first()
// => undefined
[1, 2, 3].first()
// => 1
[1, 2, 3].last()
// => 3
```

```
['salut', 'les', 'grenouilles'].indexOf('les')
// => 1
['salut', 'les', 'grenouilles'].indexOf('LES')
// => -1
```

Transformations : clear, compact, flatten, without, reverse, reduce, uniq

Pour finir, on dispose de cinq méthodes transformant le contenu du tableau. Certaines produisent un nouveau tableau (résultat `tableau` dans la syntaxe), d'autres modifient le tableau d'origine.

SYNTAXE

```
tableau.clear() -> leMemeTableauMaisVide
tableau.compact() -> tableau
tableau.flatten() -> tableau
tableau.without(obj…) -> tableau
tableau.reverse([inline = true]) -> leMemeOuUnAutre
tableau.reduce() -> tableau | valeur
tableau.uniq() -> tableau
```

La méthode `clear()` vide le tableau en ramenant sa taille à zéro, et renvoie le tableau lui-même.

La méthode `compact()` crée un nouveau tableau équivalent à l'original dont on aurait retiré tous les éléments `null` et `undefined`.

Très sympathique, la méthode `flatten()`, est abondamment utilisée en interne par Prototype : elle « aplatit » un tableau de tableaux, quel qu'en soit le niveau de profondeur. On obtient l'équivalent d'un parcours en profondeur du tableau d'origine. Nous allons l'illustrer dans les exemples.

La méthode `without` est un filtre bien pratique, qui produit un nouveau tableau équivalent à l'original purgé des arguments passés.

La méthode `reverse` a deux comportements, suivant qu'on lui passe expressément `false` ou non. Dans le premier cas, elle produit un nouveau tableau qui est l'inversion (ordre inverse) de l'original. Dans tous les autres, elle inverse directement l'original et le renvoie.

La méthode `reduce` réduit un tableau n'ayant qu'un élément à cet élément lui-même, et un tableau vide à `undefined`.

Enfin, la méthode `uniq` renvoie le tableau débarrassé de ses doublons. Le tableau n'a pas besoin d'être trié, et son ordre est inchangé.

Voici quelques exemples de ces méthodes de transformation :

Listing 4-10 Transformations de tableaux

```
var simple = [ 42, 'salut', NaN, 'les', null, 'grenouilles' ];
var complexe = [ 42, [ 'salut', [ NaN ], 'les' ], null,
    [[[ 'grenouilles' ]]] ];

simple.reverse();
// => simple : [ 'grenouilles', null, 'les', NaN, 'salut', 42 ]
simple.reverse(false)
// => [ 42, 'salut', NaN, 'les', null, 'grenouilles' ]
// => simple : [ 'grenouilles', null, 'les', NaN, 'salut', 42 ]

var simple2 = complexe.flatten();
// => simple2 : [ 42, 'salut', NaN, 'les', null, 'grenouilles' ]

simple2 = simple2.without(NaN, 'les').compact();
// => simple2 : [ 42, 'salut', NaN, 'grenouilles' ]

simple2.clear();
// => simple2 : []

[].reduce()     // => undefined
[1].reduce()    // => 1
[1, 2].reduce() // => [1, 2]

[1, 2, 3, 7, 2, 5, 7, 4, 8].uniq() // => [1, 2, 3, 7, 5, 4, 8]
```

Notez d'abord que le `without` n'a pas purgé le `NaN`, ce qui nous rappelle que par définition, `NaN` n'est égal à aucun `Number`, y compris lui-même ! Or, `without` utilise l'opérateur ==.

Extraire les éléments ayant une classe précise

Parfois, on souhaite simplement récupérer tous les éléments ayant, parmi leurs classes CSS, une classe précise. Pour ce genre d'extraction, la fonction `$$` est inutilement lourde. Prototype fournit quelque chose de plus rapide :

SYNTAXE

```
document.getElementsByClassName(className, [scope = document.body]) ->
[eltÉtendu…]
```

On précise une (et une seule) classe CSS. On récupère un tableau des éléments correspondants, garantis étendus. Si vous souhaitez restreindre la recherche à l'intérieur d'un élément particulier, précisez l'élément en deuxième argument.

Deux exemples fictifs d'invocation :

```
var hilited = document.getElementsByClassName('hilite');
var menuAlternatives = document.getElementsByClassName(
    'alternative', 'menuBar');
// On suppose qu'on a ici un conteneur d'ID 'menuBar'
```

Modules et objets génériques

À présent que nous avons vu les extensions que Prototype apporte aux objets natifs de JavaScript, il est temps d'explorer les objets, modules et espaces de noms existant hors des objets natifs, en commençant par ceux répondant à des besoins tellement courants qu'ils constituent une sorte de bibliothèque standard générique.

Nous commencerons par l'excellent module `Enumerable`, qui fournit de nombreux comportements d'extraction à tout objet fournissant une méthode d'itération spécifique. Nous poursuivrons par `Hash`, `ObjectRange`, `PeriodicalExecuter`, `Template` et, pour finir, `Try`.

Enumerable, ce héros

`Enumerable` est un module, au sens défini en début de chapitre. Il s'agit d'un ensemble de méthodes injectables dans un type existant (au travers de l'extension de son prototype), qui ne formulent que quelques dépendances sur ce type pour pouvoir fonctionner.

Avec `Enumerable`, la dépendance est simple : il faut que le type, qui est censé être un conteneur de données, dispose d'une méthode `_each` acceptant un itérateur, et que cette méthode passe à l'itérateur en question toutes les données, tour à tour. Sur cette base simple, `Enumerable` construit les nombreuses méthodes décrites ci-dessous.

« Injecter » `Enumerable` à un type satisfaisant cette contrainte est simple avec Prototype :

```
Object.extend(LeTypeQuiVaBien.prototype, Enumerable);
```

C'est exactement ce qui se passe, en interne, pour `Array`, `ObjectRange`, `Element.ClassNames` ou encore les objets `Hash` renvoyés par `$H`.

Comme le module est vaste, j'ai là aussi classé les méthodes par thème.

L'itération elle-même : each

SYNTAXE

```
objetEnumerable.each(iterateur)
```

Pour itérer sur un `Enumerable`, on utilise toujours la méthode `each` publique, jamais la méthode `_each` privée, déclarée par le type concret sur lequel on a injecté le module `Enumerable`. En effet, c'est la version publique qui prend en charge les exceptions globales `$break` et `$throw` étudiées plus haut dans ce chapitre. C'est donc grâce à elle qu'il est possible de court-circuiter une itération de ce type.

Par ailleurs, `each` passe en réalité deux arguments à l'itérateur : la valeur elle-même et la position dans le conteneur (comme d'habitude, le premier élément est à la position zéro). Par conséquent, toutes les méthodes acceptant un itérateur dans ce module ont le même comportement.

Voici deux exemples simples d'utilisation :

```
['salut', 'les', 'grenouilles' ].each(function(s) { alert(s); });
var sum = 0;
$R(1, 20).each(function(n) { sum += n; });
// sum == 210, somme des entiers de 1 à 20 inclus
```

Tests sur le contenu : all, any, include, member

Les méthodes `any` et `all` sont de plus en plus fréquentes dans les bibliothèques de conteneurs ; on les trouve d'ailleurs dans les tableaux natifs pour les navigateurs Mozilla, et il n'est pas exclu que JavaScript 2.0, le prochain standard, les officialise.

SYNTAXE

```
objetEnumerable.all([iterator]) -> booléen
objetEnumerable.any([iterator]) -> booléen
objetEnumerable.include/member([iterator]) -> booléen
```

Il s'agit de prédicats permettant de vérifier si tous les éléments d'un conteneur satisfont une condition, ou si au moins un d'entre eux la satisfait. C'est extrêmement pratique dans de nombreux cas, par exemple pour vérifier qu'un tableau ne contient que des nombres, ou que dans une liste censée contenir des éléments du DOM, il y a au moins un élément qui n'est ni `null`, ni `undefined`.

Les deux méthodes acceptent un itérateur optionnel, qu'on fournit généralement. S'il est manquant, on utilise la fonction identité, `Prototype.K`. Les éléments eux-mêmes sont alors traités comme des équivalents booléens (voir le chapitre 2 pour les correspondances).

L'itération est bien sûr court-circuitée dès que son résultat est déterminé (`all` court-circuite au premier élément insatisfaisant, `any` au premier satisfaisant).

Voici l'expression des deux tests cités en exemple un peu plus tôt :

```
if (data.all(function(x) { return 'number' == typeof x; }))
...
if (data.any())
```

Notez l'absence d'itérateur au deuxième appel : on teste directement les éléments. Avouez que c'est concis !

Il existe également une méthode `include` (et son alias `member`, c'est au choix), qui permet de tester qu'un conteneur contient bien un objet particulier. La méthode renvoie un booléen et court-circuite évidemment l'itération dès qu'elle a trouvé.

```
$R(1, 20).include(12) // => true
$A(document.childNodes).include(document.body) // => false
```

Extractions : detect, find, findAll, select, grep, max, min, pluck, reject

Les objets énumérables disposent de toute une panoplie d'extractions toutes plus pratiques les unes que les autres.

SYNTAXE

```
objetEnumerable.detect/find(iterateur) -> objet
objetEnumerable.findAll/select(iterateur) -> [objet…]
objetEnumerable.grep(pattern[, iterateur]) -> [objet…]
objetEnumerable.max([iterateur]) -> objet
objetEnumerable.min([iterateur]) -> objet
objetEnumerable.pluck(nomPropriete) -> [valeurPropriete…]
objetEnumerable.reject(iterateur) -> [objet…]
```

La méthode `detect` (et son alias `find`) recherche le premier objet satisfaisant l'itérateur dans l'énumération et le renvoie immédiatement (s'il n'a pas été trouvé, elle renvoie `undefined`). Vous voulez la première valeur supérieure à 10 dans un tableau de nombres ? Il suffit d'écrire :

```
desNombres.find(function(n) { return n > 10; })
```

La méthode `select` (et son alias `findAll`) ne se contente pas de la première valeur, mais renvoie l'ensemble des valeurs satisfaisant l'itérateur. À ce titre, elle est l'opposée de `reject`, qui renvoie toutes les valeurs ne satisfaisant pas l'itérateur.

Voici comment récupérer tous les textes de plus de 42 caractères, et comment virer les NaN :

```
desTextes.select(function(s) { return s.length > 42; })
desNombres.reject(function(n) { return isNaN(n); })
```

La méthode grep est une variante particulière de select, qui applique une expression rationnelle à la représentation textuelle (méthode toString(), pas inspect() !) des objets de l'énumération. Elle renvoie un tableau de ceux qui satisfont l'expression passée (de préférence sous forme d'un objet RegExp). Si un itérateur est précisé en second argument, il est utilisé pour récupérer une valeur dérivée de l'objet courant plutôt que l'objet lui-même.

Pour récupérer les textes ne contenant aucune voyelle, ou les longueurs de ceux contenant au moins trois mots, on procéderait ainsi :

```
desTextes.grep(/^[^aeiouy]*$/i)
desTextes.grep(/(\w+(\b|\W+)){3,}/, function(s) {
    return s.length;
})
```

Les méthodes min et max parlent d'elles-mêmes. Sans itérateur, elles travaillent sur les objets directement, qui doivent donc prendre en charge les opérateurs relationnels > et <. Un itérateur optionnel permet de rechercher le minimum et le maximum d'une valeur dérivée plutôt que des objets eux-mêmes.

Par exemple, voici comment connaître le premier texte d'une série dans l'ordre lexicographique (celui de la table des caractères), ainsi que la taille du texte le plus long :

```
desTextes.min()
desTextes.max(function(s) { return s.length; })
```

Avec un peu d'astuce, on peut obtenir par exemple le plus grand nombre de voyelles dans les textes :

```
desTextes.max(function(s) {
    return (s.match(/[aeiouy]/ig) || []).length;
});
```

Si vous comprenez cet exemple, vous pouvez commencer à aller explorer le code source de Prototype sans trop lutter...

Pour finir, la méthode pluck permet de récupérer une propriété particulière pour chaque objet de l'énumération. Cette méthode est très pratique, et largement utilisée

en interne par Prototype. Vous voulez un tableau des tailles de tous les textes ? Il suffit d'écrire :

```
desTextes.pluck('length')
```

Transformations et calculs : collect, map, inject, invoke, partition, sortBy

La frontière entre extraction et transformation est parfois ténue. Si vous estimez qu'une de ces méthodes aurait davantage trouvé sa place à la précédente section, ou réciproquement, vous avez peut-être raison. Tout dépend, en réalité, de l'utilisation concrète, et les possibilités de combinaison ou d'usage… détourné… sont potentiellement illimitées.

SYNTAXE

```
objetEnumerable.collect/map(iterateur) -> tableau
objetEnumerable.inject(accumulateur, iterateur) -> valeur
objetEnumerable.invoke(nomMethode[, arg…]) -> tableau
objetEnumerable.partition([iterateur]) -> [objVrais, objFaux]
objetEnumerable.sortBy(iterateur) -> tableau
objetEnumerable.zip(equivalentTableau…[, iterateur]) -> tableau
```

La méthode `map` (et son alias `collect`) crée un nouveau tableau où chaque élément est le résultat de l'application de l'itérateur à son homologue du tableau original. Lorsqu'il s'agit simplement de remplacer un objet par une de ses propriétés, préférez `pluck`, tellement plus simple. Mais pour quelque chose de plus avancé, `map` est incontournable. Vous voulez la parité de tous les nombres d'un tableau ? Il faut écrire :

```
[1, 2, 5, 8, 14].map(function(n) { return 0 == n % 2; })
// => [false, true, false, true, true]
```

La méthode `inject` est très utile (nous nous en sommes d'ailleurs déjà servis dans des exemples précédents, pour `Number.toColorPart()` par exemple) : elle permet de construire une valeur à partir de l'ensemble des objets de l'énumération.

On utilise une variable dite accumulateur, qui part d'une valeur initiale fournie en premier argument. Ensuite, à chaque valeur de l'itération, l'itérateur est invoqué avec la valeur actuelle de l'accumulateur comme premier argument, et l'objet courant en second. L'itérateur doit renvoyer la nouvelle valeur de l'accumulateur.

La fonction `inject` renvoie la valeur finale de l'accumulateur. Il peut s'agir d'une chaîne de caractères, d'un nombre, d'un booléen… Tout dépend de vos besoins !

Voici par exemple comment établir la somme et le produit internes d'un tableau de nombres :

```
desNombres.inject(0, function(acc, n) { return acc + n; })
desNombres.inject(1, function(acc, n) { return acc * n; })
```

Notez l'importance de la valeur initiale pour l'accumulateur. Je précise que l'itérateur prend en troisième argument la position de l'objet courant dans le tableau, même si on s'en sert plutôt rarement.

La méthode `invoke` n'est pas sans rapport à `pluck`. Là où `pluck` récupère une propriété, `invoke` appelle pour chaque objet de l'énumération la méthode désignée, en lui passant les éventuels arguments supplémentaires, et fournit le tableau des résultats. Si vous souhaitez par exemple obtenir les 10 premiers caractères de chaque texte, vous pourriez utiliser le `map` classique :

```
desTextes.map(function(s) { return s.substring(0, 10); })
```

Ou préférer `invoke`, plus élégant et pas plus lent pour un sou :

```
desTextes.invoke('substring', 0, 10);
```

La méthode `partition` sépare le bon grain de l'ivraie, en produisant deux tableaux : celui des objets ayant satisfait l'itérateur (s'il n'est pas fourni, on utilise l'équivalence booléenne des objets eux-mêmes), et celui des autres. C'est une sorte de combinaison entre `select` et `reject`, mais c'est plus efficace que d'appeler les deux séparément.

```
var result = desTextes.partition(function(s) {
    return s.length > 10;
});
// result[0] contient les textes assez longs
// result[1] contient les textes trop courts
```

La méthode `sortBy` permet de produire une variante triée du tableau, suivant un critère défini par l'itérateur obligatoire (pour trier les objets directement, on dispose déjà de la méthode native `sort` sur `Array`). Par exemple, on peut vouloir trier des textes par taille :

```
desTextes.sortBy(function(s) { return s.length; })
```

La dernière méthode de cette section, `zip`, est assez complexe. Il s'agit de « coller » des tableaux les uns aux autres, côte à côte si l'on peut dire. La méthode produit dans tous les cas un nouveau tableau résultat.

Imaginons par exemple que vous ayez un tableau d'identifiants utilisateur, et un autre de mots de passe. Vous souhaitez produire un tableau de paires : identifiant + mot de passe. Pour cela, vous avez plusieurs moyens. Si un tableau de tableaux vous convient (tableaux internes à deux éléments), alors c'est tout à fait trivial :

```
var logins = [ 'tdd', 'al', 'nioute', 'doudou' ];
var passwords = [ 'blah', 'koolik', 'linux', 'lachef', 'toto' ];
var auth = logins.zip(passwords);
// => [[ 'tdd', 'blah' ], [ 'al', 'koolik'],
//     [ 'nioute', 'linux' ], [ 'doudou', 'lachef']]
```

Notez que les éléments surnuméraires des tableaux accolés sont ignorés : l'itération prend le tableau appelant comme base.

Peut-être cette structure en tableaux imbriqués vous pose-t-elle problème, et vous préféreriez des objets avec des propriétés nommées. Qu'à cela ne tienne ! Vous pouvez préciser comme dernier argument un itérateur chargé de construire la valeur accolée sur la base du tableau interne initialement prévu. Voici notre appel précédent ajusté :

```
var auth = logins.zip(passwords, function(args) {
    return { login: args[0], password: args[1] };
});
alert(auth[0].login + ' / ' + auth[0].password);
// => 'tdd / blah'
```

Notez qu'il est ainsi possible d'accoler autant de tableaux qu'on le souhaite. Si vous aviez par exemple un tableau supplémentaire de *salts* pour l'encryptage des mots de passe, vous pourriez tout à fait l'ajouter :

```
var auth = logins.zip(passwords, salts, function(args) {
    return { login: args[0], password: args[1], salt: args[2] };
});
```

Conversions : toArray, entries, inspect

Enfin, il existe quelques conversions simples sur les objets énumérables.

SYNTAXE

```
objetEnumerable.toArray/entries() -> tableau
objetEnumerable.inspect() -> chaine
```

D'abord tout objet énumérable peut être converti en tableau, par exemple avec `$A`, puisqu'on dispose d'une méthode `toArray()` (et d'un alias, `entries`). Notez que sur

un Array, toArray() renvoie donc un clone du tableau d'origine, ce qui peut s'avérer pratique de temps à autre.

Par ailleurs, les objets énumérables fournissent bien sûr une méthode inspect(), qui renvoie une représentation texte orientée débogage. Pour information, il s'agit en réalité de celle disponible dans Array, encadrée de '<#Enumerable:' et de '>'. Notez que lorsqu'un objet incorporant Enumerable redéfinit sa méthode inspect() (comme c'est le cas pour Array et Hash, notamment), la version redéfinie a priorité.

Tableaux associatifs avec Hash

Un Hash est une sorte de tableau associatif entre des clés et des valeurs. En JavaScript, tout objet peut être considéré comme un Hash, puisqu'il ne s'agit finalement que d'une série de propriétés définies comme des paires nom + valeur (bon, il y a aussi le prototype, c'est vrai).

L'objet Hash de Prototype fournit quelques méthodes conçues pour traiter l'objet enrobé (car on construit souvent un Hash comme enrobage d'un objet existant) expressément comme une série de paires clé + valeur.

Il faut aussi savoir qu'un Hash obtenu par $H incorpore Enumerable (et ce n'est pas rien !). L'itérateur reçoit alors comme objet courant une paire nom + valeur sous un format confortable : on peut s'en servir comme un tableau à deux éléments, ou utiliser ses propriétés key et value.

SYNTAXE

```
hash.inspect() -> chaine
hash.keys() -> tableau
hash.merge(autreHash) -> hash
hash.toQueryString() -> chaine
hash.values() -> tableau
```

Les méthodes keys() et values() renvoient chacune un tableau. Bien entendu, la première renvoie les clés (si on a enrobé un objet, ce sont les noms des propriétés), tandis que la seconde renvoie les valeurs. La première est généralement beaucoup plus utilisée que la seconde. En effet, quand on a le nom d'une propriété de obj dans une variable key, la manière la plus directe d'accéder à la valeur est obj[key] !

Autre point notable : keys() ignore délibérément les méthodes de l'objet, contrairement à la traditionnelle boucle for (var key in obj), qui ne fait pas de détail entre les méthodes et les champs.

Par exemple, si vous voulez connaître tous les champs accessibles pour l'objet `navigator` du DOM niveau 0, il suffit de lancer ceci :

```
alert($H(navigator).keys().sort().join('\n'));
```

Inutile de s'appesantir sur la méthode `inspect()`, au rôle désormais classique. En revanche, la méthode `merge` est pratique : elle permet de combiner deux `Hash` pour produire un nouveau `Hash` résultat. Ce dernier contient toutes les clés des deux `Hash` d'origine. En cas de clé dupliquée, la valeur du `Hash` passé en argument aura écrasé celle du `Hash` d'origine.

La dernière méthode, `toQueryString()`, s'avère fort utile lorsque nos scripts doivent composer des paramètres de requête `GET`, ou un corps URL-encodé de requête `POST`. En effet, cette méthode construit une représentation textuelle URL encodée des paires clé + valeur du `Hash`. En construisant sa série de paramètres dans un `Hash` ou un objet anonyme, on peut donc aisément obtenir le texte final (ce qui est plus simple que de le confectionner soi-même). Petit exemple rapide :

```
var params = { title: 'Un Exemple', age: 28 };
var encoded = $H(params).toQueryString();
// => title=Un%20Exemple&age=28
```

Remarquez qu'il s'agit là de la fonction réciproque de `String.parseQuery`. Nous nous en servirons dans nos prochains chapitres sur Ajax.

ObjectRange : intervalles d'objets

Il s'agit d'un petit objet simple, qui vise à encapsuler une définition de boucle, généralement numérique. Toutefois, il peut fonctionner sur n'importe quel objet implémentant une opération `succ()` pour passer d'une valeur à la suivante.

SYNTAXE

```
$R(debut, fin[, finExclue = false]) -> intervalle
intervalle.include(valeur) -> booléen
```

La seule manière propre de créer un `ObjectRange` est d'utiliser la fonction globale `$R`, présentée plus haut dans ce chapitre.

En revanche, la méthode `include` exige que les objets sous-jacents prennent en charge les opérateurs relationnels < et <=. Ceci s'entend au sens dynamique de JavaScript, pas au sens statique de compilation des *templates* C++. Par exemple, si on n'appelle pas la méthode `include`, le fait que les objets sous-jacents n'implémentent pas un comportement cohérent sur < et <= est sans importance.

`ObjectRange` incorpore bien entendu `Enumerable`, ce qui fait qu'il permet non seulement de réaliser des itérations, mais aussi de bénéficier de toutes les opérations fournies par le module.

Des exemples d'utilisation ont été présentés à plusieurs reprises dans ce chapitre.

PeriodicalExecuter ne se lasse jamais

S'il vous arrive de devoir exécuter une fonction à intervalles réguliers, vous pouvez utiliser `PeriodicalExecuter`. Très honnêtement, cet objet n'a qu'une utilité : il évite de rentrer de nouveau dans la méthode si son invocation précédente n'est pas terminée.

SYNTAXE

```
new PeriodicalExecuter(callback, intervalInSeconds)
pe.stop()
```

Pour stopper la répétition, il suffit d'appeler la méthode `stop`. L'objet se passant comme premier argument de la fonction de rappel, celle-ci peut décider d'être la dernière invocation en appelant `stop` sur l'argument.

Attention ! Si vous passez une méthode, pensez conserver son lien (binding) correctement.

Exemples d'utilisation :

```
new PeriodicalExecuter(function() { window.title += '.'; }, 1);
...
var notifier = $('notifier');
new PeriodicalExecuter(notifier.cleanup.bind(notifier), 5);
```

Vous devriez réévaluer vos modèles

La version 1.5.0 a vu apparaître un petit objet curieux : `Template`. Cet objet permet de gérer des syntaxes de substitution au sein d'un modèle textuel.

SYNTAXE

```
new Template(templateText[, pattern]) -> modele
modele.evaluate(scopeObject) -> chaine
```

Par exemple, si vous examinez la méthode `String.gsub`, exposée plus haut dans ce chapitre, vous voyez une syntaxe de type `#{propriété}`, avec gestion d'un *backslash* devant le `#` pour désamorcer l'interprétation. Ce travail est réalisé en interne par un objet `Template`. Je vais appeler de tels objets des **modèles**.

Un modèle est défini par deux caractéristiques :

1. Son motif textuel, qui constitue le patron sur lequel confectionner le texte résultat à chaque évaluation.
2. Son motif de détection, qui est une expression rationnelle décrivant les portions dynamiques du motif textuel. Si vous souhaitez utiliser le motif par défaut, comme pour `String.gsub` par exemple, vous n'avez pas besoin de passer ce second argument à la construction : le motif par défaut `Template.Pattern` est alors utilisé.

Le principe est cependant toujours le même : le motif de détection vise à isoler un nom de propriété (en l'occurrence un champ, pas une méthode) qui sera récupérée dans l'objet de contexte fourni à l'évaluation. On peut en effet réutiliser un même modèle autant de fois qu'on veut, en appelant sa méthode `evaluate`, à laquelle on fournit l'objet contenant les propriétés à substituer dans le corps du texte.

Voyez l'exemple suivant :

```
var person = { name: 'Élodie', age: 25, height: 165 };
var tpl = new Template('#{name} a #{age} ans et mesure #{height}cm.');
alert(tpl.evaluate(person));
// => 'Élodie a 25 ans et mesure 165cm.'
```

N'est-ce pas sympathique en diable ?

Utiliser un motif de détection personnalisé

Toutefois, supposons à présent que vous deviez travailler sur les éléments d'un tableau, et que cette syntaxe vous semble trop lourde. Vous pouvez fournir votre propre motif de détection. Seule condition (mais de taille pour les débutants en expressions rationnelles), celui-ci doit isoler trois groupes :

1. Le caractère ou l'ancre situé juste avant la portion à substituer.
2. La portion à substituer complète.
3. La partie de la portion à substituer qui constitue le nom de la propriété à récupérer dans l'objet de contexte.

Par exemple, le motif par défaut, `Template.Pattern`, a l'aspect suivant :

```
/(^|.|\r|\n)(#\{(.*?)\})/
```

On distingue bien les trois groupes :

1. `(^|.|\r|\n)` peut correspondre au début de texte, à un retour chariot ou saut de ligne, ou un autre caractère quelconque.
2. `(#\{(.*?)\})` correspond à tout le bloc à substituer, du # initial à l'accolade fermante.

3 `(.*?)` correspond au nom de la propriété, constitué d'un nombre quelconque de caractères sauf l'accolade fermante (le `*?` est un *quantificateur réticent*, qui s'arrête le plus tôt possible).

Le premier groupe est nécessaire pour permettre à `evaluate` d'ignorer les portions précédées d'un *backslash*. Le second groupe lui permet justement de laisser le texte original tel quel dans un tel cas. Le troisième groupe lui donne le nom de la propriété à aller chercher.

Imaginons donc que nous souhaitons une syntaxe à base de dollars : `$index`. On considère que les indices sont constitués exclusivement de chiffres arabes. Notez qu'une telle syntaxe, sans caractère délimiteur de fin, empêche de coller de tels chiffres immédiatement à la suite de nos substitutions (ce qui tend à démontrer l'utilité de tels encadrements). Voici notre motif de détection, très inspiré de celui par défaut :

```
/(^|.|\r|\n)(\$(\d+))/
```

Voyons le résultat :

```
var data = [ 'Élodie', 25, 1.65 ];
var tpl = new Template('$0 a $1 ans et mesure $2m.',
/(^|.|\r|\n)(\$(\d+))/);
alert(tpl.evaluate(data));
// => 'Élodie a 25 ans et mesure 1.65 m.'
```

Notez que si la propriété est introuvable, `evaluate` substitue la chaîne vide (`''`).

Try.these cherche une méthode valide

Pour terminer avec les nouveaux objets à utilisation assez générique, voici l'objet `Try`, qui n'existe pour l'instant que comme espace de noms pour la méthode `these`. L'avantage étant que « `Try.these` », c'est très lisible.

Cette merveilleuse petite méthode reçoit une série de fonctions en arguments. Elle va les appeler l'une après l'autre jusqu'à en rencontrer une qui ne lève aucune exception. Elle renvoie alors son résultat.

On utilise `Try.these` lorsqu'on doit créer un code portable qui est obligé de tester plusieurs implémentations connues d'un même comportement. Ce type de situation devrait disparaître peu à peu, au fur et à mesure que les navigateurs s'aligneront sur toujours plus de standards, mais on rencontre encore quelques cas difficiles.

Dans Prototype, l'exemple roi est celui qui consiste à obtenir un objet `XMLHttpRequest`, cas de figure que nous allons étudier dès le prochain chapitre. Suivant le navigateur, on ne procède pas de la même manière. Sur MSIE 5.x, on récupère un objet ActiveX nommé `Microsoft.XMLHTTP`. Sur MSIE 6, le nom a changé :

`Msxml2.XMLHTTP`. Sur tous les autres navigateurs (y compris MSIE 7, qui s'est enfin aligné sur le standard émergeant), on crée directement un objet natif JavaScript.

En temps normal, gérer ce genre d'alternatives donne des imbrications verbeuses de blocs `try/catch`. Avec `Try.these`, tout s'éclaire. Voyez plutôt le code de `Ajax.getTransport()`, dans Prototype :

Listing 4-11 Ajax.getTransport(), l'exemple roi de Try.these

```
return Try.these(
    function() {return new XMLHttpRequest()},
    function() {return new ActiveXObject('Msxml2.XMLHTTP')},
    function() {return new ActiveXObject('Microsoft.XMLHTTP')}
) || false;
```

Remarquez qu'on passe trois fonctions en arguments. `Try.these` tentera les trois, dans cet ordre. La première qui n'échoue pas en levant une exception verra sa valeur de retour utilisée. Si aucune ne fonctionne, `Try.these` renvoie `undefined`, ce qui permet ci-dessus de basculer sur `false`.

Manipulation d'éléments

Nous en avons terminé avec les nouveaux objets « génériques » de Prototype. Il s'agit principalement de structures de données et de petits objets utilitaires au niveau algorithmique. Attaquons maintenant des objets dédiés à la manipulation du contenu de la page.

Nous verrons d'abord les manipulations valables pour tout élément du document, avec `Element`. Après un rapide détour par `Selector` (sans lequel la fonction globale `$$` n'existerait pas), nous verrons les objets dédiés aux formulaires, à la gestion unifiée des événements, et enfin aux ajouts dynamiques de contenu.

Element, votre nouveau meilleur ami

L'objet `Element` est un espace de noms pour de nombreuses méthodes destinées à manipuler des éléments du DOM. En raison de leur nombre, je les ai classées par thème pour s'y retrouver plus facilement.

Element.Methods et les éléments étendus

Techniquement, ces méthodes sont en réalité définies dans un module `Element.Methods`, lequel est ensuite incorporé à `Element`. La raison de cette séparation, qui peut d'abord sembler superflue, réside dans la notion d'élément étendu, que nous avons déjà abordée en début de chapitre.

En isolant les méthodes d'extension des quelques méthodes « administratives » internes à `Element`, Prototype est capable de les injecter si nécessaire dans les éléments du DOM qui vous sont retournés par des fonctions comme `$` ou `$$`.

Le résultat net : dans la vaste majorité des cas, les deux syntaxes suivantes sont strictement équivalentes :

SYNTAXE

```
Element.methode(votreElement, arg…)
votreElement.methode(arg…)
```

Quelques petits exemples pour fixer les idées :

```
Element.hide('monId');
$('monId').hide();
monElement.hide();
Element.update(monElement, '<strong>Génial !</strong>');
$('monId').update('<strong>Génial !</strong>');
monElement.update('<strong>Génial !</strong>');
```

Quand ça ne veut pas...

Il reste une question en suspens : dans quels cas cet accès direct (`monElement.methode`) n'est-il pas disponible ? Il faut que les conditions suivantes soient réunies :

1. Votre navigateur ne propose pas d'objet prototype pour les éléments issus du DOM. Vous pouvez tester cela en chargeant `prototype.js` dans une page et en examinant la variable globale privée `_nativeExtensions` : si elle est à `false`, vous êtes dans ce cas.
2. Vous travaillez sur un élément que vous n'avez pas récupéré au travers d'une méthode documentée dans ce chapitre comme renvoyant un élément garanti étendu (`eltÉtendu` dans les blocs de syntaxe).

À l'heure où j'écris ces lignes, Firefox, Camino, Mozilla, Safari et Konqueror proposent un objet prototype pour les éléments du DOM. La condition n°1 est donc remplie uniquement sur MSIE !

Le principal cas de figure où la condition n°2 est remplie est la traversée « à l'ancienne » du DOM. Vous récupérez par exemple un élément avec `$('sonId')`, puis vous commencez à naviguer avec `firstChild`, `nextSibling`, `parentNode`, etc. Cette navigation ne passe bien sûr pas par Prototype, et si la condition n°1 est remplie, les éléments retournés ne sont pas étendus.

Autre cas courant remplissant la condition n°2 : la récupération de l'élément cible d'un événement avec `Event.element`, ou d'un élément dérivé avec `Event.findElement`.

Nous verrons ces méthodes plus tard dans ce chapitre. Notez qu'il suffit d'enrober l'appel dans la fonction `$` pour retrouver les extensions.

Dans la pratique, ceci n'est pas un problème. Vous connaissez votre code, et si vous utilisez une traversée du DOM, vous êtes au courant. Dans un tel contexte, vous n'avez qu'à utiliser la version indirecte (`Element.methode(elt)`) des méthodes, qui est d'ailleurs celle qu'on trouve sur la plupart des articles et didacticiels, pour la simple et bonne raison que la version directe n'est apparue qu'avec la version 1.5.0.

Le reste du temps, vous pouvez normalement utiliser sans crainte la version directe, qui est en effet plus concise, plus élégante, et globalement plus naturelle.

Néanmoins, par souci de garantie de fonctionnement, les sections qui suivent documenteront les syntaxes indirectes. À vous de garder à l'esprit qu'une version directe est le plus souvent disponible.

Valeur de retour des méthodes

Toutes les méthodes de `Element` (ainsi que de `Form` et `Form.Element`) qui modifient l'élément visé renvoient cet élément. Cela permet d'enchaîner les modifications, par exemple :

```
$('sidebar').addClassName('selected').show();
```

Au total, cela concerne tout de même 27 méthodes...

Élément es-tu là ? hide, show, toggle et visible

Quatre méthodes gèrent la visibilité de l'élément. Elles utilisent toutes la propriété CSS `display`, qu'elles placent à `'none'` pour masquer l'élément, et qu'elles désactivent (en la ramenant à `''`) pour afficher l'élément. Ceci suppose deux choses :

1. Les éléments concernés seront retirés du flux du document lorsqu'ils sont masqués, engendrant un *reflow* s'ils faisaient partie du flux (un élément positionné de façon absolue ou fixe n'en fait pas partie, par exemple).
2. Si vous souhaitez masquer l'élément d'entrée de jeu, vous devez le faire avec un attribut style dans le document, et non avec la feuille CSS. C'est le seul cas de figure dans ce livre qui justifie une légère intrusion de l'aspect dans le contenu.

SYNTAXE

```
Element.hide(elt) -> elt
Element.show(elt) -> elt
Element.toggle(elt) -> elt
Element.visible(elt) -> booléen
```

Les noms me semblent parfaitement explicites. Les méthodes `hide` et `show` masquent l'élément en exigeant un `display` à `'none'` ou en retirant cette surcharge, tandis que `toggle` bascule de l'un à l'autre. Les trois acceptent autant d'éléments que vous le souhaitez.

La méthode `visible` est en réalité un prédicat, qui indique si l'élément est visible ou non. Elle le considère invisible si sa propriété `display` est à `'none'`. Il est dommage que Prototype ne préfixe pas ses méthodes prédicats par un `'is'`...

Gestion du contenu : cleanWhitespace, empty, remove, replace et update

Prototype fournit plusieurs méthodes visant à modifier ou remplacer le contenu d'un élément, voire à remplacer l'élément lui-même, ou carrément le retirer du DOM.

SYNTAXE

```
Element.cleanWhitespace(elt) -> elt
Element.empty(elt) -> booléen
Element.remove(elt) -> elt
Element.replace(elt, html) -> elt
Element.update(elt, html) -> elt
```

La méthode `cleanWhitespace` supprime tout nœud fils textuel vide (constitué uniquement de *whitespace*). C'est une sorte d'épuration du DOM qui permet de simplifier les algorithmes de traversée (`firstChild`, `lastChild` ou `nextSibling` sont généralement des éléments comme on pourrait s'y attendre, plutôt que des nœuds textuels vides dûs à l'indentation d'un code source).

La méthode `empty` est un prédicat indiquant si l'élément est vide, c'est-à-dire ne contenant au plus que du *whitespace*.

Comme son nom le laisse supposer, `remove` retire l'élément du DOM. C'est juste une simplification du sinueux `elt.parentNode.removeChild(elt)`. Mais en accès direct, avouez que `elt.remove()`, c'est plus sympatique !

Les deux méthodes `replace` et `update` ne diffèrent que sur un point : `update` remplace le contenu de l'élément, tandis que `replace` remplace l'élément lui-même ! La différence est cependant de taille pour vos algorithmes. Dans la pratique, `update` est plus fréquemment utilisée.

Les deux interprètent le bloc HTML comme suit : les fragments de script sont retirés avant insertion dans le DOM, puis ces mêmes scripts sont exécutés à part, juste après l'insertion (à 10 ms d'écart, vous ne risquez pas de sentir la différence). L'idée est que ces fragments de script ne sont pas censés persister dans le DOM, mais qu'il peut être intéressant d'injecter tant du contenu que du comportement dans un document (notamment pour les réponses Ajax).

Styles et classes : addClassName, classNames, getElementsByClassName, getElementsBySelector, getStyle, hasClassName, match, removeClassName et setStyle

Nous disposons de nombreuses méthodes, et même d'un objet dédié, pour gérer les classes CSS et les propriétés CSS individuelles.

SYNTAXE

```
Element.addClassName(elt, className) -> elt
Element.classNames(elt) -> Element.ClassNames -> elt
Element.getElementsByClassName(elt, className) -> [eltÉtendu…]
Element.getElementsBySelector(elt, règle…) -> [eltÉtendu…]
Element.getStyle(elt, style) -> valeur
Element.hasClassName(elt, className) -> booléen
Element.match(elt, singleSelector) -> booléen
Element.removeClassName(elt, className) -> elt
Element.setStyle(elt, { prop1: val1[,…] }) -> elt
```

Comme vous le savez, un même élément peut avoir plusieurs classes CSS associées, en séparant leurs noms par des espaces. Afin de vous éviter d'avoir à gérer les manipulations de texte, vous disposez donc des méthodes `addClassName`, `removeClassName` et `hasClassName`. On peut également examiner l'ensemble des classes d'un élément en appelant la méthode `classNames`, qui renvoie un objet dédié `Element.ClassNames` (voir plus bas).

Vous pouvez par ailleurs manipuler confortablement les propriétés CSS individuelles de votre élément au travers des méthodes `getStyle` et `setStyle`. Derrière leur apparente simplicité, elles réalisent un travail complexe.

Ainsi, `getStyle` ne cherche pas qu'à vous renvoyer la valeur spécifiée de la propriété. Si une telle valeur existe, elle vous la renvoie bien entendu. Dans le cas contraire, elle se débrouille avec les possibilités du navigateur (DOM niveau 2 Style ou API propriétaire) pour extraire la valeur calculée de la propriété, qui peut par exemple résulter d'un style par défaut ou de l'héritage. Si la propriété a la valeur spéciale `'auto'`, `getStyle` renvoie `null`. Sinon, elle renvoie la valeur. Enfin, notez que le nom de propriété passé à `getStyle` peut être celui de la recommandation CSS ou celui de la propriété DOM correspondante. Voir l'exemple de `String.camelize`, plus haut dans ce chapitre, pour un rappel.

Si ces notions de valeur spécifiée, valeur calculée, cascade ou héritage sont floues pour vous, n'hésitez pas à lire tranquillement le début de l'annexe B, qui les décrit avec précision.

La méthode `setStyle` utilise une syntaxe plus évoluée pour son deuxième argument, afin de permettre la définition d'un nombre quelconque de propriétés en un seul

appel. On lui passe un objet anonyme dont les propriétés ont le nom (CSS ou DOM) de la propriété CSS voulue, avec leurs valeurs. Exemples :

```
Element.setStyle('indicator', { opacity: '0.6' });
$('notif').setStyle({ background: '#fdd', color: 'maroon'});
```

La méthode `match` accepte un sélecteur unique (donc sans espaces) et renvoie `true` si l'élément satisfait ce sélecteur.

Enfin, la méthode `getElementsByClassName` est un équivalent local de la méthode homonyme dans document, et `getElementsBySelector` est un équivalent local de la fonction `$$`.

L'objet Element.ClassNames

L'objet `Element.ClassNames` est utilisé pour gérer les classes CSS d'un élément. Employé en interne par des méthodes comme `addClassName` ou `hasClassName`, il est également retourné pour utilisation directe par la méthode `classNames`. Notez qu'il incorpore `Enumerable`.

SYNTAXE

```
ecn.set(className)
ecn.add(className)
ecn.remove(className)
ecn.toString() -> chaine
```

Les noms des méthodes parlent d'eux-mêmes. `add` et `remove` ajoutent ou retirent à la liste courante de classes, tandis que `toString()` renvoie la représentation de l'attribut `class` correspondant : les noms des classes séparés par des espaces. Si vous trouvez que des méthodes manquent, souvenez-vous qu'on dispose de toutes les méthodes de `Enumerable`, par exemple `include` !

Enfin, `set` remplace toute la liste par une nouvelle définition, dont la syntaxe est celle de l'attribut HTML `class` (et de l'attribut DOM `className`) : les noms séparés par des espaces.

Les copains d'abord : ancestors, descendants, nextSiblings, previousSiblings, siblings

Disponibles depuis la 1.5.0 RC1, ces méthodes permettent de récupérer les nœuds autour de l'élément.

SYNTAXE

```
Element.ancestors(elt) -> [eltÉtendu…]
Element.descendants(elt) -> [elt…]
Element.nextSiblings(elt) -> [eltÉtendu…]
Element.previousSiblings(elt) -> [eltÉtendu…]
Element.siblings(elt) -> [eltÉtendu…]
```

La lignée (éléments ancêtres) est renvoyée par `ancestors` depuis le nœud père jusqu'au plus lointain ancêtre. Le tableau renvoyé par `descendants` suit l'ordre du document. Les méthodes `nextSiblings` et `siblings` suivent l'ordre du document, tandis que `previousSiblings` suit l'ordre inverse (en s'éloignant de l'élément d'origine).

Petite incohérence, `descendants` reste pour le moment la seule à ne pas renvoyer des éléments garantis étendus. Si cela vous embête pour la généricité de vos traitements, sachez qu'il suffit de transformer son retour comme ceci :

```
var descs = $('eltId').descendants().map(Element.extend);
```

Bougez ! down, next, previous et up

La RC1 a introduit quatre nouvelles méthodes destinées à se déplacer très rapidement dans les éléments du DOM.

SYNTAXE

```
Element.down(elt[, expression][, index]) -> [eltÉtendu…]
Element.next(elt[, expression][, index]) -> [eltÉtendu…]
Element.previous(elt[, expression][, index]) -> [eltÉtendu…]
Element.up(elt[, expression][, index]) -> [eltÉtendu…]
```

Le sens de déplacement est évident ; par exemple, `down` utilise les éléments fils, tandis que `previous` passe par `previousSibling`. Les éléments renvoyés le sont dans l'ordre prévu par les méthodes correspondantes dans la section précédente.

L'argument optionnel `expression` fournit un sélecteur CSS unique (sans espaces), tandis que `index` permet d'indiquer la position de la correspondance sur laquelle on souhaite arriver (démarre à zéro). On peut utiliser l'un, l'autre, ou les deux.

Voici un exemple tiré du journal des modifications de Prototype, pour vous aider à comprendre. Supposons l'arborescence suivante :

```
<div id="sidebar">
    <ul id="nav">
        <li>…</li>
        <li>…</li>
        <li class="selected">…</li>
```

```
    </ul>
    <ul id="menu">
        …
```

Voici maintenant quelques utilisations et leurs résultats commentés.

Tableau 4–1 Invocations de up, down, previous et next

Code	Élément résultat	Commentaires
`$('nav').up()` `$('menu').up('div')`	`<div id="sidebar">`	Montée d'un cran sans contrainte aucune, ou jusqu'au premier ancêtre `div`.

`$('sidebar').down()`
`$('sidebar').down('ul')`
`$('menu').previous()`
	`<ul id="nav">`	Descente d'un cran depuis `sidebar`, descente jusqu'à un `ul`, ou élément précédent `menu`…

`$('sidebar').down(1)` `$('sidebar').down('li')`	Premier `<li>…</li>`	Descente de deux crans (position 1, on démarre à zéro) depuis `sidebar`, ou descente jusqu'à un `li`.

`$('sidebar').down(2)`
`$('sidebar').down('li', 2)`
`$('sidebar').down('li').next('li')`
	Deuxième `<li>…</li>`	Descente de trois crans depuis `sidebar`, voire avec contrainte `li` en plus, ou prochain `li` après le premier `li` descendant de `sidebar`.

`$('sidebar').down('li.selected')`	`<li class="selected">`
	`…`
	`</li>`
		Descente jusqu'au premier `li` ayant une classe CSS `selected`.

`$('sidebar').down('ul').next()`	`<ul id="menu">`	Élément suivant le premier `ul` descendant de `sidebar`.

Remarquez notamment que `next` et `previous` éliminent le problème des nœuds texte vides issus de l'indentation, que nous avons détaillé au chapitre 3.

Positionnement : getDimensions, getHeight, makePositioned, undoPositioned

Quelques méthodes contrôlent le positionnement. Il ne s'agit pas tant d'une API totale (par exemple, on n'a pas de `getWidth`, qui serait pourtant triviale à écrire) que de méthodes couvrant certains besoins de Prototype. La plupart du temps, vous utiliserez plutôt l'objet `Position`.

SYNTAXE

```
Element.getHeight(elt) -> Number
Element.getDimensions(elt) -> { width: Number, height: Number }
Element.makePositioned(elt) -> elt
Element.undoPositioned(elt) -> elt
```

La méthode `getHeight` renvoie la hauteur en pixels de l'élément. La méthode `getDimensions()` est plus complète, en renvoyant un objet doté de deux propriétés : `width` et `height`, deux nombres exprimés en pixels.

`makePositioned` examine la propriété CSS `position` de l'élément : si elle est indéfinie, ou explicitement spécifiée à `static` (sa valeur par défaut), elle la passe en `relative` après avoir sauvegardé son ancienne valeur pour restauration.

Cette restauration est justement l'affaire de `undoPositioned`, qui remet la propriété CSS `position` à son ancienne valeur. Cette méthode n'a donc de sens que suite à un `makePositioned`.

Ces quatre méthodes sont abondamment utilisées en interne par des objets comme `Position`, et sont également très utiles à la bibliothèque script.aculo.us, en particulier pour ses effets visuels et la gestion du glisser-déplacer.

Défilement et troncature : makeClipping, scrollTo, undoClipping

Voici trois méthodes gérant le défilement et la troncature de contenu.

SYNTAXE

```
Element.scrollTo(elt) -> elt
Element.makeClipping(elt) -> elt
Element.undoClipping(elt) -> elt
```

La méthode `scrollTo` fait si besoin défiler la vue du navigateur de façon à ce que l'élément y soit visible. C'est très pratique suite à l'insertion dynamique d'un élément par exemple, pour s'assurer que l'internaute voit bien la modification. Et cela nous évite de devoir recourir à des bricolages navrants d'ancre dans l'URL de la page.

Les méthodes `makeClipping` et `undoClipping` ont la même relation que `makePositioned` et `undoPositioned`. Il s'agit ici de déterminer si, lorsqu'une boîte dimensionnée a trop de contenu, celui-ci doit étirer son conteneur ou être partiellement masqué, la boîte conservant sa taille.

Techniquement, `makeClipping` sauvegarde la valeur actuelle de la propriété CSS `overflow`, et la passe à `hidden`. `undoClipping` la restaure.

Dans tous les cas, sachez que la version actuelle de Konqueror (à l'heure où j'écris ces lignes, la version 3.5.2) ne prend pas totalement en charge la propriété CSS `overflow`, utilisée par ces méthodes de troncature. Les résultats sont donc surprenants, comme vous pourrez le voir sur les exemples de l'archive des codes source du livre.

childOf le mal nommé

Pour en finir avec `Element`, il reste une dernière méthode, `childOf`, qui est toutefois légèrement mal nommée (un peu comme tous les arguments `frequency` de Prototype, qui sont en réalité des périodes, et devraient donc se nommer au minimum `interval`...).

En effet, en termes DOM, un élément fils est situé directement sous son élément père. Un élément situé à n'importe quel niveau de profondeur est ce qu'on appelle un élément descendant.

SYNTAXE

```
Element.childOf(elt, ancestor) -> booléen
```

Or, c'est précisément ce que fait `childOf` : elle teste si l'élément est un descendant de l'élément indiqué en deuxième argument. Elle aurait sans doute dû s'appeler `descendantOf`... Mais je vous l'accorde, je chipote un peu.

Selector, l'objet classieux

Voici l'objet utilisé, en interne, par la fonction globale `$$`. Il permet de représenter une série contiguë de sélecteurs CSS :

- Élément (nom de la balise)
- ID (`#id`)
- Attribut
  - `[attr]`
  - `[attr="value"]`
  - `[attr!="value"]`
  - `[attr~="value"]`
  - `[attr|="value"]`
- Classe (`.laClasse`)

En revanche, le sélecteur de descendants, l'espace (`' '`), est traité directement par la fonction `$$`. Un `Selector` ne peut donc pas le gérer. Les sélecteurs de fils (>) et d'adjacence (+) ne sont pas encore pris en charge non plus.

Le seul intérêt de manipuler `Selector` consiste à vouloir « cacher » une analyse de règle CSS (sans espaces) pour utilisation massive.

SYNTAXE

```
new Selector(expr)
sel.findElements([scope = document]) -> [eltÉtendu…]
sel.toString() -> chaine
```

Pour créer manuellement un tel `Selector`, on utilise tout simplement `new Selector(expression)`. L'expression est analysée et l'objet initialisé, prêt à l'emploi.

Par la suite, on peut réaliser l'extraction correspondante en appelant sa méthode `findElements`. Celle-ci prend un argument optionnel `scope`, qui précise la portion du document dans laquelle chercher. C'est justement grâce à cet argument que `$$` implémente le sélecteur de descendants. Si vous ne le précisez pas (ce qui devrait être le cas général), on utilise tout le document. On récupère un tableau d'éléments étendus.

Enfin, la méthode `toString()` renvoie une représentation textuelle de l'expression CSS correspondant au sélecteur. Dans une utilisation manuelle, l'intérêt est mineur : c'est vous qui avez fourni l'expression...

Manipulation de formulaires

Prototype fournit de nombreux objets dédiés à la manipulation des formulaires et de leurs champs. On trouve d'abord `Field` et `Form`, qui travaillent respectivement aux niveaux du champ et du formulaire entier.

Certaines classes, dites observateurs, réagissent aux événements de modification. `Form.Observer` réagit à toute modification dans un champ quelconque du formulaire ; il se repose en fait sur une série d'objets `Form.Element.Observer`.

Field / Form.Element

L'objet `Form.Element` (et son alias `Field`) fournit des méthodes de manipulation d'un ou plusieurs champs de formulaires. On fournit comme d'habitude soit l'ID soit une référence directe à l'élément.

J'insiste : l'ID, pas le nom de champ (attribut `id`, pas `name`) !

SYNTAXE

```
Field.activate(elt) -> elt
Field.clear(elt) -> elt
Field.disable(elt) -> elt
Field.enable(elt) -> elt
Field.focus(elt) -> elt
```

```
Field.getValue(elt) -> value | [value…]
Field.present(elt) -> booléen
Field.select(elt) -> elt
Field.serialize(elt) -> chaineURLEncodée
```

Les noms parlent d'eux-mêmes, à quelques précisions près.

Je rappelle d'abord que les méthodes figurant ici sont automatiquement ajoutées aux éléments étendus lorsque c'est approprié.

Commençons par les méthodes sans surprises : `clear` efface le champ passé, `focus` donne... le focus (fait du champ la cible des saisies clavier), et `select` sélectionne le texte à l'intérieur du champ (utile uniquement pour les saisies de texte : type `text`, type `password` et balise `textarea`).

La méthode `present` renvoie `true` uniquement si le champ passé a une valeur non vide (au sens strict : si elle ne contient que du *whitespace*, elle sera tout de même considérée remplie).

La méthode `activate` est une sorte de *combo* : elle appelle d'abord `focus`, et si le type de champ s'y prête, elle appelle ensuite `select`. Globalement préférable à `focus`.

La méthode `serialize` fournit une représentation URL encodée du champ, avec son nom et sa valeur. Si le champ est à valeur multiple (cas d'un `select` en mode `multiple`), il est présent autant de fois que nécessaire.

La méthode `getValue` renvoie la valeur du champ. Pour un champ à valeur simple, celle-ci est directement renvoyée. En valeur multiple, on obtient un tableau des valeurs sélectionnées.

Vous vous souvenez de la fonction globale `$F` ? C'est en réalité un alias de `Form.Element.getValue` !

Ces deux méthodes se reposent en fait lourdement sur l'objet technique interne `Form.Element.Serializers`.

L'objet `Form.Element` est aussi utilisé en interne par l'objet `Form`, qui utilise une sorte de composition/délégation pour de nombreux traitements.

Un mot sur Form.Element.Serializers

Cet objet technique contient des méthodes de routage vers le bon traitement d'extraction de valeur. Ses méthodes renvoient en réalité un tableau à deux éléments : le nom du champ et sa valeur.

Je ne rentrerai pas dans les détails, mais voici l'essentiel de son comportement :

- Pour un champ `input` de type `submit`, `hidden`, `password`, `search` ou `text`, ainsi que pour un champ `textarea`, la valeur est utilisée telle quelle, dans tous les cas.

- Pour un champ `input` de type `checkbox` ou `radio`, on ne renvoie quelque chose que si le champ est coché.
- Pour un champ `select` simple (pas d'attribut `multiple="multiple"`), on récupère l'option sélectionnée. S'il n'y en a pas, la valeur est la chaîne vide (`''`). Si l'option sélectionnée n'a pas d'attribut `value`, son texte est utilisé à la place, conformément à la recommandation W3C.
- Pour un champ `select` multiple, la valeur est un tableau des valeurs sélectionnées, chacune étant déterminée comme ci-dessus (attribut `value` si présent, texte sinon).

Form

L'objet `Form` permet de manipuler un formulaire dans sa globalité. L'argument `form` est bien sûr soit l'ID soit la référence directe de l'élément `form`.

SYNTAXE

```
Form.disable(form)
Form.enable(form)
Form.findFirstElement(form) -> elt
Form.focusFirstElement(form)
Form.getElements(form) -> [elt…]
Form.getInputs(form[, typeName][, name]) -> [elt…]
Form.reset(form)
Form.serialize(form) -> chaineURLEncodée
```

Je rappelle d'abord que les méthodes figurant ici sont automatiquement ajoutées aux éléments `form` étendus.

Commençons par les méthodes simples : `disable` et `enable` désactivent et réactivent l'ensemble des champs du formulaire. Un champ qui avait le focus le perdra juste avant de se désactiver. La liste des champs est obtenue par `getElements`. Pratique si l'envoi manuel du formulaire donne un traitement Ajax pendant lequel un second envoi serait problématique.

La méthode `reset()` se contente de réinitialiser le formulaire, comme le bouton obtenu par `<input type="reset" />`. Je rappelle qu'il s'agit simplement de ramener les champs à la valeur qui leur est donnée dans le HTML.

La méthode `getElements` renvoie un tableau de tous les éléments constituant des champs du formulaire, classés par balise : d'abord les `input`, puis les `textarea` et enfin les `select`. Cet ordre peut amener à des comportements inattendus de `findFirstElement` ou `focusFirstElement`.

En effet, `findFirstElement` devrait renvoyer le tout premier élément non caché (c'est-à-dire qui ne soit pas un `input` de type `hidden`) et actif (non désactivé, si vous

préférez) du formulaire. Cela signifie intuitivement « le premier dans l'ordre du document ». Or, si votre premier élément est un `select` par exemple, et qu'il est suivi par des `input`, c'est le premier `input` qui sera sélectionné (en supposant que tous soient actifs et visibles).

La méthode `focusFirstElement` est une combinaison de confort : elle appelle `activate` sur le résultat de `findFirstElement`, tout simplement.

Personnellement, je ne suis pas un grand fan de `findFirstElement`. Un formulaire n'a pas forcément comme premier champ activé le premier champ visible (si celui-ci apparaît logiquement en premier, mais dispose d'une valeur par défaut le plus souvent satisfaisante, par exemple). L'ordre de tabulation n'est pas forcément celui du document. Et bien sûr, le premier champ activé peut parfaitement devoir être un `select` ou un `textarea` (ce deuxième cas est un peu plus rare).

Je préfère passer par `$$` pour obtenir mon premier champ actif et l'activer manuellement, généralement comme ceci (supposons que mon formulaire ait pour ID `'mainForm'`) :

```
$$('#mainForm *[tabindex=1]')[0].activate()
```

Après, les goûts et les couleurs...

La méthode `getInputs` est plus spécifique que `getElements`. Elle ne s'occupe que des éléments `input`, et permet de filtrer le résultat sur la base du type ou du nom (ou les deux). Imaginons que vous ayez des boutons radio : ceux représentant les variantes d'une même donnée auront le même nom de champ. Vous pouvez les obtenir comme ceci :

```
Form.getInputs('mainForm', null, 'newsletterMode')
```

Ou même plus spécifique, pour plus de sécurité :

```
Form.getInputs('mainForm', 'radio', 'newsletterMode')
```

À moins que seules les cases à cocher, quelles qu'elles soient, vous intéressent :

```
Form.getInputs('mainForm', 'checkbox')
```

Enfin, la méthode `serialize` renvoie la représentation URL encodée du formulaire, comme on peut avoir besoin de la transmettre comme paramètres `GET` ou corps `POST` dans une requête Ajax. Elle se repose sur la méthode homonyme de `Form.Element`, qu'elle appelle pour chaque élément en assemblant les résultats.

Form.Observer

Voici notre premier exemple d'observateur. Dans Prototype, un observateur est un mécanisme surveillant à intervalle régulier la valeur d'un élément. Lorsque celle-ci change (ainsi qu'à la première observation, qui n'a pas encore de valeur précédente à comparer), l'observateur appelle une fonction de rappel qu'on lui a fourni à la construction.

SYNTAXE

```
new Form.Observer(form, intervalInSecs, callback)
```

Techniquement, les observateurs sont définis par l'objet `AbstractObserver`, qui est étendu par divers objets, dont `Form.Observer`. Ces objets ont juste besoin d'implémenter une méthode `getValue()`, qui renvoie la valeur ainsi observée.

Un observateur a toujours trois arguments dans son constructeur :

1. L'élément observé (ici un formulaire).
2. L'intervalle (ou la période, pour les mordus) en secondes.
3. La fonction de rappel. Celle-ci recevra deux arguments : l'élément observé, et sa nouvelle valeur.

Afin de prendre en compte l'ensemble de ses champs, un `Form.Observer` définit le résultat de `Form.serialize` comme étant sa valeur.

Voici un exemple d'utilisation (qui semble bien pénible pour l'internaute) :

```
function formChanged(form, newValue) {
    alert($H(newValue.parseQuery()).toArray().join('\n'));
}
new Form.Observer('mainForm', 1, formChanged);
```

On peut imaginer utiliser ce genre de chose pour soumettre les modifications à la volée en Ajax, par exemple. Si c'est pour de la complétion automatique de texte, je vous conseille toutefois plutôt d'utiliser l'objet `Ajax.Autocompleter` de script.aculo.us. Une merveille, et déjà tout prêt !

Form.Element.Observer

Cet objet permet de réagir à intervalle régulier au changement d'un seul champ plutôt que de n'importe quel champ du formulaire.

SYNTAXE

```
new Form.Element.Observer(elt, intervalInSecs, callback)
```

La valeur est bien entendu obtenue par `Form.Element.getValue`... Cet observateur s'utilise comme les autres (voir l'exemple donné pour `Form.Observer`).

Gestion unifiée des événements

Je l'ai signalé à de multiples reprises au chapitre précédent, Prototype brille particulièrement par sa capacité à unifier la gestion des événements. On l'a vu, tous les navigateurs répandus ne se conforment pas toujours exactement au DOM niveau 2 événements, et MSIE est particulièrement dans l'erreur, en adoptant une approche entièrement distincte du standard. Cette situation constitue souvent un casse-tête pour les développeurs web souhaitant mettre en œuvre des traitements événementiels un tant soit peu étoffés.

Au travers de son objet `Event`, Prototype nous fournit tout le nécessaire pour associer ou révoquer des gestionnaires d'événements, et analyser les détails d'un événement lorsqu'il survient.

Event

L'objet `Event` sert d'espace de noms pour les méthodes liées à la gestion événementielle. Considérez que l'objet passé à vos gestionnaires d'événements pour représenter l'événement est propriétaire au navigateur. Les méthodes de `Event` sont là pour s'y retrouver entre les implémentations ; elles offrent une sorte d'API unifiée au-dessus des objets propriétaires.

SYNTAXE

```
Event.KEY_(BACKSPACE|DELETE|DOWN|END|ESC|HOME|LEFT|PAGEUP|PAGEDOWN|
         ➥ RETURN|RIGHT|TAB|UP)
Event.element(evt) -> Element
Event.findElement(evt, tagName) -> Element
Event.isLeftClick(evt) -> booléen
Event.observe(elt, evtName, observer[, useCapture = false])
Event.pointerX(evt) -> Number
Event.pointerY(evt) -> Number
Event.stop(evt)
Event.stopObserving(elt, evtName, observer[, useCapture = false])
```

Le premier argument de chaque méthode est l'objet qui est passé, lui aussi, en premier argument à vos gestionnaires d'événements. Cet objet vient du navigateur, et sa nature exacte varie. Prototype se débrouille.

Pister un événement

Pour associer un gestionnaire à un événement sur un élément précis, on utilise `Event.observe`. On précise l'élément, le nom DOM de l'événement (pas de préfixe `'on'`), le gestionnaire à utiliser, et éventuellement l'utilisation du mode capture (voir le chapitre 3 pour plus d'informations sur la capture).

> **ATTENTION ! Le binding de vos gestionnaires**
>
> Si votre gestionnaire est une méthode qui a besoin d'accéder aux champs d'instance de son objet, n'oubliez pas d'utiliser `bindAsEventListener`. Souvenez-vous que `bind` oublierait quant à elle de vous transmettre l'objet événement en premier argument.

Voici un premier exemple avec une fonction classique, qui cache un élément dès qu'on clique dessus :

Listing 4-12 Inscription d'une fonction comme gestionnaire d'événement

```
function hideElement(event) {
    Element.hide(Event.element(event));
}

Event.observe('grandTimide', 'click', hideElement);
```

Nous verrons la méthode `Event.element` dans quelques instants.

Voici à présent une autre version, qui garde un compteur de clics pour chaque élément cliqué (ce qui permet d'utiliser le même gestionnaire pour de nombreux éléments, on suppose toutefois qu'ils ont tous des ID), et le fait disparaître au bout de 3 clics.

Comme le gestionnaire est une méthode qui a besoin de pouvoir accéder aux champs de son objet, on utilise `bindAsEventListener` :

Listing 4-13 Inscription d'une méthode comme gestionnaire d'événement

```
var DelayedHider = {
    _countKeeper: $H(),
    handleClick: function(event) {
        var elt = Event.element(event);
        var count = this._countKeeper[elt.id] || 0;
        this._countKeeper[elt.id] = ++count;
        if (3 == count)
            Element.hide(elt);
    }
};
Event.observe('petitTimide', 'click',
    DelayedHider.handleClick.bindAsEventListener(DelayedHider));
```

C'est l'occasion de retrouver notre bon vieux `||` pour gérer les valeurs par défaut, ou comme ici le cas initial, lorsque le compteur n'existe pas dans le `Hash`.

On peut bien sûr vouloir révoquer un gestionnaire, c'est-à-dire cesser de réagir à un événement particulier, pour un élément particulier. La révocation utilise très exactement les mêmes arguments que l'inscription, mais la méthode s'appelle, très logiquement, `stopObserving`.

BOGUE À la poursuite des fuites de mémoire

On a constaté que sur MSIE 6, les gestionnaires inscrits causaient des fuites de mémoire au déchargement de la page (navigation vers une autre URL, etc.) : tout n'était pas correctement libéré. Prototype 1.5.0 corrige ce souci en tenant à jour une sorte de registre de tous les gestionnaires inscrits par `observe` et non encore révoqués par `stopObserving`. Au déchargement de la page, tous les gestionnaires encore inscrits sont explicitement révoqués. Hop ! Envolées, les fuites de mémoire !

Démasquer l'élément qui a reçu l'événement

Lorsqu'un événement survient, votre gestionnaire se réveille. Mais l'élément qui a reçu l'événement n'est pas forcément celui sur lequel votre gestionnaire est inscrit.

En vertu du principe de bouillonnement, il peut s'agir d'un élément descendant : ainsi, un gestionnaire `click` pour un paragraphe peut se déclencher si on clique sur un texte dans un élément `strong` à l'intérieur de ce paragraphe...

Par ailleurs, vous pouvez parfaitement avoir utilisé un même gestionnaire pour de multiples éléments, comme on vient de le voir au listing 4-13. Alors comment distinguer ? Peste ! Mais comme au listing 4-13, justement : avec `Event.element`.

Cette méthode prend l'objet événement reçu et se débrouille pour extraire une référence à l'élément cible. Attention, l'élément n'est pas étendu. C'est d'ailleurs pourquoi, tout à l'heure, j'utilisais `Element.hide(elt)` au lieu de `elt.hide()`. Vous souvenez-vous que je l'avais déjà signalé en creusant la notion d'élément étendu, plus haut dans ce chapitre ?

Il existe également une méthode `Event.findElement`. Elle permet de récupérer le plus proche élément ancêtre de l'élément « cible » pour une balise donnée. Par exemple, vous souhaitez récupérer le plus proche `div` conteneur de votre élément cible. Peut-être ce dernier est-il un `span` calé dans le coin supérieur droit de ce div, jouant le rôle d'une case de fermeture... Plutôt que de naviguer à la main dans le DOM, utilisez :

```
var containerDiv = Event.findElement(event, 'div');
```

Je recommande une casse minuscule, comme toujours (optique XHTML), mais dans la pratique `findElement`, dans sa grande mansuétude, s'en moque. Si aucun élément ancêtre avec ce nom de balise n'est trouvé, il renvoie `null`.

Étouffer la propagation de l'événement

La plupart du temps, vos gestionnaires sont les seuls censés devoir traiter un événement particulier (par exemple un clic spécifique, ou l'envoi d'un formulaire). Dans de tels cas, vous pouvez vouloir stopper la propagation de l'événement.

Peut-être votre gestionnaire implémente-t-il un contrôle validant le comportement par défaut de l'événement (on pense principalement aux envois de formulaire). Si votre algorithme le décide, il faut pouvoir annuler ce comportement par défaut.

Ces deux opérations sont indissociables avec Prototype : il considère que si vous annulez le comportement par défaut d'un événement, le propager risquerait d'induire d'autres gestionnaires en erreur, tandis que si vous annulez la propagation, c'est probablement que le comportement par défaut ne vous intéresse pas.

Et dans la pratique, il a raison. Si des contre-exemples vous viennent à l'esprit, commencez par réviser votre DOM niveau 2 événements pour vérifier que les événements auxquels vous pensez se propagent effectivement, ou que leur comportement par défaut est bien celui que vous croyez.

La méthode utilisée est `Event.stop`. On lui passe l'objet événement, et elle interrompt sa propagation tout en annulant son comportement par défaut. C'est très pratique.

Déterminer l'arme du crime : souris ou clavier

Savoir que l'événement s'est produit, et sur quel élément il a originalement eu lieu, ne suffit pas toujours. Loin de là ! Lorsque tous les navigateurs implémentent de façon uniforme une information, Prototype ne rajoute rien. En revanche, lorsqu'on a des disparités, il fournit une API unifiée.

L'état de la souris

Les boutons de la souris sont représentés de façon assez diversifiée d'un navigateur à l'autre, sans parler de la plate-forme. Prototype se borne à vous dire si le clic est considéré clic gauche classique, avec `Event.isLeftClick` (n'oubliez pas que sur le Mac, par exemple, on n'a pas de clic droit ; on utilise généralement Ctrl+Clic).

La position du curseur est un concept à géométrie variable. La grande question est en effet : dans quel référentiel ? La page ? La portion visible de la page ? L'écran ?

Le DOM niveau 2 événements prévoit deux couples de propriétés pour l'objet événement que vous recevez :

- `screenX` et `screenY` : position à l'écran ;
- `clientX` et `clientY` : position dans la partie visible de la page ;

Prototype prévoit par ailleurs les besoins de positionnement et de glisser-déplacer en définissant la notion de position dans la page, au travers des deux méthodes `Event.pointerX` et `Event.pointerY`.

Les touches du clavier

On trouve une série de constantes `KEY_`*xxx* qui représentent les principales touches spéciales du clavier. De nouvelles constantes sont ajoutées au fil des versions de Prototype, selon les besoins apparus entre temps, et après vérification de l'unicité de la valeur sur toutes les plates-formes.

Les touches correspondant à des caractères ASCII n'ont pas besoin de constantes, car leur valeur correspond au caractère recherché. Si vous préférez, une constante `KEY_8` serait égale au code du caractère `'8'`. Je rappelle que BackSpace correspond à la touche française Ret. Arr., Delete à notre Suppr, Home à notre Orig et End à notre Fin.

Sur quoi utiliser ces constantes et valeurs, me direz-vous ? Eh bien, il se trouve que les principaux navigateurs implémentent de façon homogène l'information, pour une fois. Utilisez la propriété `keyCode` de l'objet événement que votre gestionnaire reçoit, comme ceci :

```
function handleKeyPress(event) {
    if (KEY_RETURN == event.keyCode)
        // code de traitement
}
```

Les événements clavier et souris sont par ailleurs normalement dotés de propriétés booléennes indiquant l'état d'enfoncement des principaux modificateurs : Alt, Ctrl et Maj. Ce sont les propriétés `altKey`, `ctrlKey` et `shiftKey`, respectivement. Vous pouvez donc facilement réagir à un Ctrl+G :

```
function handleKeyDown(event) {
    if ('G'.charCodeAt(0) == event.keyCode && event.ctrlKey)
        // Code de traitement
}
```

Le `charCodeAt(0)` est nécessaire car `'G'` est une `String`, par un `char` (qui n'existe pas en JavaScript). L'expression `'G' == 71` donne `false`. Il faut donc obtenir le code numérique du premier caractère de cette `String`.

Form.EventObserver

`Form.EventObserver` est similaire à `Form.Observer` : il s'agit de déterminer si un champ du formulaire a changé de valeur depuis le dernier examen, et le cas échéant, d'appeler une fonction de rappel.

SYNTAXE

```
new Form.EventObserver(form, callback)
```

Mais au lieu d'examiner périodiquement le formulaire, on réagit aux événements. En l'occurrence, un `Form.EventObserver` enregistre un gestionnaire pour tous les champs du formulaire (événement `click` pour les cases à cocher et boutons radio, événement `change` pour les autres). C'est donc plus immédiat que `Form.Observer`, ce qui n'est pas forcément un mieux, suivant l'ergonomie recherchée.

Form.Element.EventObserver

On retrouve la relation `Form` / `Form.Element` déjà vue pour les observateurs basés sur intervalle. Il s'agit ici de ne réagir qu'à l'événement de modification d'un seul champ.

SYNTAXE

```
new Form.Element.EventObserver(elt, callback)
```

Insertions dynamiques

Voici notre dernière section. Après cela, nous aurons fait le tour complet de Prototype 1.5.0, à l'exception de `Position`, que j'ai trouvé bien trop complexe, et bien trop destiné à des bibliothèques tierces plutôt qu'à du code « développeur de site », pour être traité ici.

L'insertion dynamique de contenu est potentiellement complexe à réaliser manuellement (surtout de façon portable !), et Prototype fait des merveilles.

Les insertions sont basées sur un objet `Abstract.Insertion`, spécialisé pour chaque type d'insertion. C'est un schéma courant dans Prototype, où les observateurs spécialisent aussi un objet « abstrait », par exemple. Une insertion se crée toujours avec deux arguments : l'objet de référence, et le (X)HTML représentant le contenu à insérer. Le type d'insertion est défini par l'objet utilisé.

SYNTAXE

```
new Insertion.Before(elt, content)
new Insertion.Top(elt, content)
new Insertion.Bottom(elt, content)
new Insertion.After(elt, content)
```

Toute insertion procède de la même manière : le contenu sans ses éventuels scripts est inséré, et immédiatement après (10 millisecondes plus tard, pour les puristes), ces fragments de scripts sont exécutés avec `eval`. Il s'agit de la même logique que celle vue dans `Element.update` et `Element.replace`.

Les quatre insertions possibles correspondent aux quatre positions disponibles :

- `Before` insère immédiatement avant l'élément ;
- `Top` insère au début du contenu de l'élément ;
- `Bottom` insère à la fin du contenu de l'élément ;
- `After` insère immédiatement après l'élément.

L'exemple fourni dans l'archive des codes source pour ce livre illustre ces possibilités. En voici une capture d'écran après les quatre insertions.



Figure 4–2
L'exemple de cette section après quatre insertions

Pour aller plus loin...

Sites

- Le site officiel de Prototype, pour récupérer la dernière version finalisée :
  http://prototype.conio.net/
- Le site officiel de script.aculo.us, pour disposer de nombreux effets et outils tout prêts, et voir des utilisations avancées de Prototype :
  http://script.aculo.us/
- Les « fenêtres Prototype », une bibliothèque très sympathique de Sébastien Gruhier permettant de simuler des fenêtres, modales ou non, avec une gestion de thèmes (*skins*), des fonctionnalités de débogage, et j'en passe. Basée sur Prototype et script.aculo.us, cette bibliothèque ouvre des horizons ! En plus, le site fournit une documentation de bonne qualité :
  http://prototype-window.xilinus.com/

Groupe de discussion

Le groupe Google RubyOnRails-Spinoffs a été créé en août 2006 pour servir de centre d'aide technique, avec les avantages d'indexation et de recherche de Google. On y trouve de nombreux membres avec un bon niveau technique :
http://groups.google.com/group/rubyonrails-spinoffs

Canal IRC

Enfin, il faut noter qu'on trouve souvent une réponse rapide et fiable sur le canal IRC dédié à Prototype, hébergé sur l'incontournable serveur `irc.freenode.net`. Le canal se nomme tout simplement `#prototype`.

Deuxième Partie

Ajax, ou l'art de chuchoter

Jusqu'ici, nous avons examiné les arcanes de JavaScript, et celles du DOM, qui est au cœur de la manipulation dynamique des pages. Ensuite, nous avons présenté l'essentiel de la bibliothèque Prototype, qui rend nos codes JavaScript plus portables, mais surtout plus courts, plus simples, plus expressifs et tellement plus élégants.

Comprendre et maîtriser ces piliers du Web 2.0 constitue un préalable nécessaire à l'apprentissage de cet univers technologique dont Ajax est le sujet phare. Nous avons déjà étudié dans le détail le « J ». Restent les aspects « Asynchronous » et « XML ». Dans les chapitres qui suivent, nous allons découvrir puis explorer en profondeur les composantes techniques concernées.

On commencera par examiner la technologie nue, au travers de sa clé de voûte, l'objet `XMLHttpRequest`, qui réalise des requêtes HTTP asynchrones pour le compte de scripts dans la page, et nous permet ensuite de traiter la réponse, en tant que document XML si besoin est. Bien comprendre les rouages de cet objet est indispensable pour éviter le risque d'une dépendance à l'un ou l'autre des nombreux frameworks disponibles autour d'Ajax. Nous verrons que Prototype est, bien entendu, encore présent pour nous faciliter grandement la tâche, tant sur des utilisations triviales que plus avancées.

Nous découvrirons aussi quelques autres frameworks, au premier rang desquels l'incontournable script.aculo.us, de Thomas Fuchs. Non content d'avoir créé une bibliothèque portable d'effets visuels spectaculaires (lesquels, employés à bon escient, augmentent considérablement la qualité de notre ergonomie), Thomas nous offre une panoplie de solutions toutes prêtes pour les emplois les plus courants d'Ajax, comme la complétion automatique.

5

Les mains dans le cambouis avec XMLHttpRequest

L'objet `XMLHttpRequest` est responsable, à lui seul, des aspects asynchrone et XML d'Ajax. C'est lui qui va effectuer des requêtes HTTP internes, invisibles, et asynchrones (s'exécutant en arrière-plan sans bloquer l'exécution des scripts ou geler la page) à une couche serveur. C'est lui aussi qui va traiter la réponse, éventuellement en tant que document XML.

Nous allons d'abord examiner ensemble l'anatomie d'une conversation Ajax entre une page web et la couche serveur. Nous installerons ensuite une couche serveur, justement, afin de pouvoir réaliser nos tests. Après un petit rappel historique sur `XMLHttpRequest`, nous rentrerons dans les détails techniques de son utilisation, et examinerons ensuite les différents types de réponse qu'on peut renvoyer, avec leurs intérêts respectifs.

Anatomie d'une conversation Ajax

Une conversation Ajax est constituée d'un ou plusieurs échanges, généralement asynchrones, entre la page web (au travers de son code JavaScript) et une couche serveur. On verra d'ailleurs au chapitre 8 que cette couche serveur peut être extrêmement variée, et n'a en rien l'obligation de résider sur le même serveur que vos pages.

Chaque échange suit la séquence que voici :

1 Création (ou réutilisation) d'un requêteur.
2 Association d'un gestionnaire d'état (une fonction qui nous appartient et qui va notamment traiter la réponse quand celle-ci arrivera).
3 Définition de la requête : mode de synchronisation, méthode HTTP (`GET`, `POST`, etc.), URL destinataire, paramètres et données éventuels.
4 Envoi de la requête.
5 Invocations du gestionnaire d'état au fil du cycle de vie de la requête, en particulier aux stades suivants : lancement, réceptions de parties de réponse, fin de réception de réponse.
6 Lorsque la réponse a été complètement reçue, traitement de celle-ci par notre code JavaScript (par exemple, insertion d'un nouvel élément dans une liste, affichage de suggestions pour un champ de saisie, affichage d'une portion d'image supplémentaire).

Il est important de bien comprendre qu'entre les étapes 4 et 5, la page « vit sa vie », ainsi que ses codes JavaScript. Elle n'est en aucun cas bloquée. Il est toujours possible d'utiliser une requête synchrone, mais cela bloque justement la page le temps du traitement, et réduit à néant l'intérêt du système ! En asynchrone, l'utilisateur conserve la possibilité d'interagir avec elle. On verra que cet aspect, qui constitue l'avantage fondamental de la technologie, n'est toutefois pas sans dangers ergonomiques et nécessite souvent une conception intelligente de l'interface.

Dans ce chapitre, nous aurons l'occasion d'examiner et de manipuler toutes ces étapes en détail, mais il va de soi que dans une utilisation industrielle, sur des projets réels et potentiellement lourds, on ne saurait s'encombrer des détails à chaque utilisation. Les chapitres suivants nous montreront comment tirer parti de frameworks répandus pour masquer l'apparente complexité du processus et nous concentrer sur la partie fonctionnelle.

Un petit serveur pour nos tests

Afin de pouvoir tester du code Ajax, nous devons avoir mis en place une couche serveur. En termes concrets, il nous faut donc, avant toute chose, un logiciel serveur HTTP installé, configuré et en cours d'exécution. Le tout doit résider soit sur notre propre machine, soit sur une machine joignable depuis la nôtre, puisque nous utiliserons notre navigateur pour visualiser nos pages de démonstration.

Mais ce n'est pas tout. Si nous nous arrêtions là, cela impliquerait que les contenus renvoyés par ce serveur HTTP soient figés, ce qui réduit grandement l'intérêt des tests et démonstrations que nous pourrions mener.

Il faut donc aussi que ce serveur prenne en charge un langage permettant de renvoyer un contenu dynamique. Le choix qui semble évident, pour cela, est PHP, en raison de sa popularité, de l'abondance de documentation et de sa prise en charge par virtuellement tous les hébergeurs. Et pourtant, ce n'est pas le choix que nous retiendrons. Voici pourquoi :

1. Nous ne souhaitons pas lier la possibilité de mener les tests à la disponibilité d'un compte et d'un espace de stockage chez un hébergeur. L'argument de la prise en charge est donc nul.
2. Nous ne souhaitons pas lier la facilité de mise en place d'un serveur de test à la plate-forme. Bien que PHP soit facile à utiliser à des fins de test sous Windows, notamment avec des produits comme EasyPHP, il nécessite une configuration élaborée sous Linux et Mac OS X.
3. Nous souhaitons éviter les éventuels conflits de ressource entre le serveur de test et des services déjà déployés. Ainsi, si vous êtes développeur (le simple fait que vous lisiez cet ouvrage signifie probablement que vous êtes développeur web, par exemple), vous avez peut-être déjà des serveurs MySQL, IIS ou Apache installés et opérationnels sur votre machine.
4. Nous souhaitons permettre aux lecteurs n'ayant pas une expérience préalable d'un langage côté serveur de s'y retrouver sans trop de mal dans nos exemples. Le langage retenu se doit donc d'être le plus lisible possible, ce qui n'est pas vraiment le cas de PHP, en particulier pour un néophyte.
5. La configuration nécessaire à nos tests doit être la plus simple possible. En ce qui nous concerne, il s'agit simplement d'associer à certaines URL du serveur des morceaux de code aptes à générer du contenu dynamique.
6. Le lancement et l'arrêt du serveur, ainsi que le suivi de son exécution, doivent être simplifiés au maximum. Ici, les serveurs classiques, même emballés dans une interface centralisée comme EasyPHP, sont déjà trop complexes, en particulier si la configuration pose problème. Le suivi de l'exécution, quant à lui, est tout simplement absent de tels outils, nécessitant des commandes tierces comme le `tail` d'Unix, ainsi que la connaissance de l'emplacement des fichiers de journalisation.
7. Plus subjectivement, j'estime qu'à technologie de pointe, langage de pointe ! Des possibilités comme PHP, JSP ou ASP reposent sur des langages rigides ou empreints d'idées anciennes.

Pour toutes ces raisons, j'ai décidé d'utiliser Ruby.

Pas de panique ! Peut-être ce langage ne vous dit-il rien, ou peut-être avez-vous déjà aperçu quelques bribes de code Ruby qui vous auraient semblé exotiques. Toutefois, j'estime qu'il s'agit là d'un choix judicieux pour ce chapitre, à de nombreux titres. En miroir aux considérations listées précédemment, les principales raisons de mon choix sont :

1. Ruby est très facile à installer, tant sur Windows que Linux ou Mac OS X. L'installation fournit une large bibliothèque standard d'objets et de services, dont un serveur léger HTTP tout équipé, apte bien sûr à exécuter du code Ruby pour générer du contenu.
2. WEBrick, le serveur HTTP fourni dans la distribution standard de Ruby, est très facile d'emploi. Il nous suffira de quelques lignes de code très simples pour décrire sa configuration, y compris l'association entre des URL et nos fonctions.
3. Lancer le serveur se fait d'une courte saisie dans la ligne de commande. L'arrêter est l'affaire d'un simple Ctrl+C. Le suivi de l'exécution a lieu automatiquement dans la console, sans avoir à chercher de fichiers de log.
4. Puisque le serveur est sur notre propre machine, nul besoin d'un compte chez un hébergeur, ni de procédures fastidieuses de déploiement sur ce compte.
5. Ruby est un langage très facile à apprendre et à lire. Et même si l'objet de ce chapitre n'est pas de vous enseigner les rudiments du langage (il ne s'agit après tout ici que d'un outil au service de l'apprentissage des mécanismes Ajax), vous verrez que le code parle de lui-même (et quand ce n'est pas le cas, les explications nécessaires sont courtes).
6. Ruby est un langage moderne, 100 % objet et très flexible, extrêmement productif, conçu d'après les leçons tirées de la mise en œuvre des langages récents.
7. Les principaux acteurs d'Ajax, notamment les auteurs des bibliothèques Prototype et script.aculo.us, sont très impliqués dans le framework Ruby on Rails (RoR), écrit intégralement en Ruby. Les deux technologies, qui contribuent à « élargir le champ des possibles » (*push the envelope*), ne sont donc pas sans rapport.

Installation de Ruby

Commençons donc par installer le nécessaire sur votre machine ! Rassurez-vous, il ne s'agit pas ici d'installer un énorme système de développement ; rien de commun avec Visual Studio.NET ou simplement le JDK de Sun. La taille varie suivant la plateforme, mais dans tous les cas, elle est bien inférieur à 100 Mo.

Sous Windows

Un excellent projet est tout spécialement dédié aux développeurs Ruby sous Windows : l'installateur Ruby « en 1 clic ». Il s'agit d'un projet Open Source disponible sur http://rubyinstaller.rubyforge.org.

L'installateur met en place les éléments suivants :

- Une version très récente de Ruby (à l'écriture de ces lignes, la 1.8.4).
- L'éditeur libre SciTE (http://www.scintilla.org/SciTE.html), qui fournit de quoi travailler pour les développeurs souhaitant un éditeur puissant sans pour autant utiliser un EDI (environnement de développement intégré).
- L'EDI FreeRIDE (http://freeride.rubyforge.org/wiki/wiki.pl), un environnement complet pour travailler avec Ruby, qui fournit les fonctionnalités habituelles de ce type de produit : coloration syntaxique, complétion de code, modèles, exécution, débogage, etc.
- De nombreuses bibliothèques et outils Ruby couramment utilisés. On y trouve entre autres Rake, l'outil de *make* dans l'univers Ruby ; et le système de bibliothèques RubyGems, qui formalise les bibliothèques Ruby.
- La version HTML Help de l'ouvrage de référence, *Programming Ruby*, écrit par Dave Thomas et Andy Hunt, les « Pragmatic Programmers », qui ont popularisé Ruby en Occident. Attention toutefois, il s'agit de la première édition, pour Ruby 1.6. Une seconde édition, massivement améliorée et complétée pour la version 1.8, est disponible (au format papier, PDF colorisé, ou les deux). Elle est préférable si vous souhaitez découvrir Ruby plus avant : http://pragmaticprogrammer.com/titles/ruby/index.html.

Hormis Ruby lui-même, tous ces composants sont optionnels. L'installateur est téléchargeable ici : http://rubyforge.org/frs/?group_id=167. Une fois le téléchargement terminé, voici les étapes à suivre pour l'installation :

1. Lancez le programme d'installation (par exemple, `ruby184-20.exe`).
2. Choisissez Next, puis à l'écran suivant I Agree.
3. Cochez European Keyboards (active le symbole € dans l'interpréteur interactif `irb`), quant au reste, libre à vous d'installer ce que vous voulez (l'espace disque total requis variera entre 83 et 87 Mo à l'heure où j'écris ces lignes). Sélectionnez ensuite Next.
4. Pour le chemin d'installation, `C:\ruby` est très bien, mais si vous souhaitez le changer abstenez-vous de toute espace dans le chemin retenu, afin d'éviter les problèmes potentiels. Cliquez sur Next.
5. Changez le nom du groupe de programmes à quelque chose de plus générique, par exemple Ruby. Choisissez ensuite Install.

6 L'installation s'exécute. et prend un peu de temps car elle comporte de très nombreux petits fichiers.
7 En fin d'installation, choisissez Next. Décochez Show Readme et cliquez sur Finish.

Et voilà !

Jetez un œil au groupe de programmes Ruby : il propose de nombreuses ressources de documentation, le serveur de documentation des *gemmes* (paquets Ruby) et `fxri`, outil regroupant un moteur de recherche dans l'aide en ligne (outil en ligne de commande : `ri`) et l'interpréteur interactif de Ruby (outil en ligne de commande : `irb`).

Pour information, les modifications apportées aux variables d'environnement sont les suivantes :

- INPUTRC pour autoriser le symbole € dans `irb`.
- PATH inclut désormais `C:\ruby\bin` (adapté au chemin que vous avez choisi, évidemment), pour avoir accès aux nombreux programmes fournis : `ruby`, `ri`, `irb`, `rdoc`, `gem` mais aussi `iconv` et `fxri`.
- PATHEXT pour autoriser l'invocation de scripts `.rb` et `.rbw` comme des commandes classiques (ce que nous n'utiliserons pas).

Autre point à régler sur Windows XP SP2 : l'avertissement du système chaque fois qu'on voudra lancer un serveur. La première fois que ça vous arrivera, choisissez Débloquer.

Sous Linux/BSD

La plupart des distributions Linux et des BSD ont un système de paquetages, avec Ruby et les outils connexes (notamment Rake) disponibles sous forme de paquets. Entre autres distributions proposant Ruby, on trouve :

- Debian et dérivés (Ubuntu, Kubuntu, etc.) ;
- Mandriva (anciennement Mandrake) ;
- SuSE et dérivés (comme OpenSuSE) ;
- Red Hat Linux (et Fedora) ;
- Slackware (au travers de paquets préparés sur LinuxPackages.net, par exemple) ;
- FreeBSD, NetBSD et OpenBSD.

Qui plus est, la majorité des distributions Linux ont déjà Ruby installé, car un nombre croissant d'outils Linux sont réalisés en Ruby. Commencez donc par ouvrir un shell et taper la commande :

```
ruby -v
```

Si vous obtenez un numéro de version d'au moins 1.8 (par exemple « `ruby 1.8.4 (2005-12-24) [i486-linux]` ») plutôt qu'un message d'erreur du type « commande introuvable », vous n'avez aucune installation à faire. Dans le second cas, retentez tout de même votre chance avec la commande `ruby1.8` plutôt que simplement `ruby`.

Notez que pour disposer de WEBrick, la bibliothèque de serveur HTTP léger, il vous faut au moins la version 1.8.0 de Ruby, mais cette version datant d'août 2003, une distribution n'en disposant pas ferait véritablement figure de brontosaure.

Les utilisateurs de Linux n'ont normalement pas de difficulté à installer des paquets, qu'ils passent par des outils graphiques comme SynaptiK ou utilisent des outils en ligne de commande, par exemple `apt-get` ou `aptitude`. Le nom du paquet est en général tout simplement `ruby`.

Sous Mac OS X

Un installateur « 1 clic » pour OS X est en cours de création à l'heure où nous écrivons ces lignes, par la même équipe que celle de son homologue Windows.

Toutefois, Ruby est d'ores et déjà simple à installer sous Mac OS X. Après tout, un très grand nombre de développeurs Ruby, et la majorité des développeurs Rails (RoR, vous vous souvenez ?) sont sur Mac OS X.

Tout d'abord, si vous tournez sous Jaguar (10.3) ou ultérieur, Ruby est installé d'entrée de jeu. Si vous êtes encore sur Panther (10.2), vous trouverez un `.dmg` de Ruby 1.8.2 sur la page http://homepage.mac.com/discord/Ruby/ (d'ailleurs, il y en a un pour Jaguar aussi).

Depuis Puma (10.4), Ruby est très bien pris en charge sur Mac, au point que même Ruby on Rails est installé d'office !

Un mot sur le cache

Enfin, je précise que quel que soit le système d'exploitation, le cache du navigateur peut parfois empêcher le bon fonctionnement des exemples, en particulier lorsqu'on les teste les uns après les autres à un bref intervalle (les fichiers CSS et JS pour la plupart s'appellent `client.css` et `client.js`, mais ils changent d'un exemple à l'autre). L'effet de cookies mis à jour n'est pas non plus visible si on passe par le cache. L'annexe D traite en détail de la configuration du cache dans les principaux navigateurs. Effectuez cette manipulation, sous peine d'obtenir des comportements très curieux au fil des exemples...

Un petit serveur HTTP et un code dynamique simple

Voici un script Ruby qui contient tout notre premier serveur de test !

Listing 5-1 Notre premier serveur, avec deux actions distinctes

```
#! /usr/bin/env ruby ❶

require 'webrick'❷
include WEBrick

server = HTTPServer.new(:Port => 8042) ❸
server.mount_proc('/hi') do |request, response| ❹
  response.body = 'Bonjour !'
end

server.mount_proc('/time') do |request, response| ❺
  response.body = Time.now.to_s
end

trap('INT') { server.shutdown } ❻

server.start ❼
```

❶ (Linux/OS X) Permet de lancer ce script comme un exécutable normal.
❷ Déclare un besoin du module `webrick` et l'importe dans notre script.
❸ Configure un serveur HTTP sur le port 8042.
❹ Associe l'URL `/hi` à une réponse figée, « Bonjour ! ».
❺ Associe l'URL `/time` à une réponse dynamique, contenant la date et l'heure au moment de la requête.
❻ Réagira à l'interruption (`Ctrl+C`) en arrêtant le serveur.
❼ Démarre le serveur web !

Comme vous pouvez le constater, il ne faut que très peu de lignes, assez compréhensibles, pour créer de toutes pièces un serveur web, associer des contenus fixe ou dynamique à deux URL, s'assurer de pouvoir le fermer proprement, et finalement le démarrer !

Sauvegardez ce code dans un fichier nommé `hi_timed.rb`, puis ouvrez une console (ce que Windows nomme une « invite de commandes »), placez-vous dans le répertoire contenant le fichier, et tapez simplement :

```
ruby hi_timed.rb
```

Vous allez obtenir un affichage similaire à ceci :

```
[2006-06-12 23:02:25] INFO WEBrick 1.3.1
[2006-06-12 23:02:25] INFO ruby 1.8.4 (2005-12-24) [i486-linux]
[2006-06-12 23:02:25] INFO WEBrick::HTTPServer#start: pid=16951
  ➥ port=8042
```

Ouvrez à présent un navigateur et allez sur http://localhost:8042/hi. Vous devez voir apparaître le texte « Bonjour ! ». Dans la console où vous avez lancé le serveur, vous remarquez qu'un journal de requêtes est en train de se constituer, ce qui est bien pratique lors du débogage :

```
[2006-06-12 23:02:25] INFO WEBrick 1.3.1
[2006-06-12 23:02:25] INFO ruby 1.8.4 (2005-12-24) [i486-linux]
[2006-06-12 23:02:25] INFO WEBrick::HTTPServer#start: pid=16951
  ➥ port=8042
localhost.localdomain - - [12/Jun/2006:23:04:11 CEST] "GET /hi
  ➥ HTTP/1.1" 200 9
- -> /hi
[2006-06-12 23:04:12] ERROR `/favicon.ico' not found.
localhost.localdomain - - [12/Jun/2006:23:04:12 CEST] "GET /favicon.ico
  ➥ HTTP/1.1" 404 281
- -> /favicon.ico
```

Chaque requête génère au moins deux lignes : la première qui identifie la machine cliente (ici, vous-même), la date et l'heure de la requête, le type de requête effectuée (par exemple « `GET /hi HTTP/1.1` »), le code de réponse (`200` : tout va bien) et la taille du contenu renvoyé (9 octets, soit le nombre de caractères dans « bonjour ! »). La seconde ligne indique le chemin absolu, au sein du serveur, de la ressource demandée.

Vous êtes peut-être surpris de voir une demande pour `/favicon.ico`, que vous n'avez pas faite. Tout navigateur la fait après une première requête sur un site, pour tenter de récupérer une petite icône identifiant le site, qui sera alors affichée à côté de l'URL, associée à un éventuel marque-page, placée dans l'onglet de la page s'il existe, etc.

Intéressons-nous à présent à l'action dynamique que nous avons mise en place, pour l'horodatage : naviguez sur http://localhost:8042/time. Vous obtenez un résultat du type :

```
Mon Jun 12 23:09:44 CEST 2006
```

Il s'agit de la représentation textuelle par défaut d'une date en Ruby, qui suit le format standard de la RFC 2822 (format qu'on retrouve, en interne, dans les protocoles de messagerie électronique et de consultation de pages web, par exemple). Ce n'est certes pas ce qu'il y a de plus lisible, mais qu'importe : cela nous permet de montrer l'aspect dynamique. En effet, si vous rafraîchissez la page dans votre navigateur, vous voyez

que la date est mise à jour. Le petit morceau de code Ruby dans notre serveur est invoqué à chaque requête et change son résultat à chaque fois. Nous avons là de quoi tester Ajax sur des bases plus intéressantes qu'avec un simple contenu statique.

Arrêtons à présent notre petit serveur d'exemple : il nous suffit de reprendre la console et de taper `Ctrl+C`. Le serveur s'arrête en clôturant son journal de quelques lignes stoïques :

```
[2006-06-12 23:13:35] INFO going to shutdown ...
[2006-06-12 23:13:35] INFO WEBrick::HTTPServer#start done.
```

À présent que nous avons tout le nécessaire pour simuler une application côté serveur, rendant ainsi notre exploration d'Ajax plus intéressante, il nous faut faire connaissance avec le véritable moteur d'Ajax, l'objet responsable du « A » initial : le mécanisme de requêtes asynchrones.

La petite histoire de XMLHttpRequest

C'est grâce à l'objet `XMLHttpRequest` qu'une page web, par l'intermédiaire de code JavaScript, peut « discuter », en coulisses, avec des serveurs (en effet, il ne s'agit pas obligatoirement du même serveur que celui dont provient la page).

Origines et historique

Rendons à César ce qui lui appartient : dans une sphère technologique où Microsoft, depuis la sortie de MSIE 5 en 1999, n'a guère brillé par son sens de l'innovation, il faut souligner que c'est au sein de MSIE, et dès la version 5 justement, qu'est apparu `XMLHttpRequest`. Il était (tout comme dans les versions 5.5 et 6.0) fourni sous forme d'un ActiveX. La première implémentation compatible est apparue en 2002 dans Mozilla 1.0 (on le trouve donc également dans Firefox et Camino) et les autres navigateurs ont suivi le mouvement : Safari à partir de la version , Konqueror, Opera depuis la version 8.0, et même le plus modeste iCab.

Au cœur d'Ajax, `XMLHttpRequest` est tellement utile et répandu que son API, à l'origine propriétaire, est en train de faire l'objet d'une standardisation par le W3C (http://www.w3.org/TR/XMLHttpRequest/ ; une traduction française est disponible sur http://www.xul.fr/XMLHttpRequest.html) avec un deuxième jet (*working draft*) au 19 juin 2006.

Bien préparer un échange asynchrone

D'un didacticiel à l'autre, on voit plusieurs manières de préparer une requête asynchrone. Comme nous l'avons vu en début de chapitre, le processus comporte un certain nombre d'étapes. Nous allons voir ensemble le détail, en insistant sur une méthodologie qui nous semble optimale. Mais avant, voici un petit rappel de la différence significative entre MSIE 6 et versions antérieures d'une part, et les autres navigateurs d'autre part.

ActiveX versus objet natif JavaScript

Jusqu'à la création d'une implémentation compatible par Mozilla en 2002, `XMLHttpRequest` restait une technologie centrée sur MSIE. Il n'est donc pas surprenant qu'il ait été fourni sous forme d'ActiveX, technologie qui, en 1999, n'avait pas encore vu sa réputation détruite à coups d'innombrables failles de sécurité et usages abusifs.

Cependant, les autres navigateurs, qui n'implémentent pas ActiveXintentionnellement, principalement pour des raisons de sécurité justement, ont naturellement opté pour un choix plus rationnel et, techniquement, plus simple : la mise à disposition de cette fonctionnalité sous forme d'objet natif JavaScript, bien plus facile à utiliser.

Si cette forme de mise à disposition, la seule qui puisse être multinavigateur, est aujourd'hui bien retenue (elle est exigée par la standardisation en cours du W3C, et Microsoft a annoncé que MSIE 7 utiliserait cette forme également : http://blogs.msdn.com/ie/archive/2006/01/23/516393.aspx), il reste un schisme entre les deux modes de mise à disposition, fossé qui n'est pas près de se résorber : on sait déjà que MSIE 7 ne sera disponible que sur des Windows XP SP2 à la licence vérifiée, ce qui constitue une partie plutôt minoritaire du parc Windows déployé. Les entreprises, notamment, sont souvent restées sur Windows 2000, voire Windows Me ou Windows 98. Pour tous ces postes, ainsi que les XP, XP SP1 et XP SP2 illégaux (très répandus, par exemple, sur le marché asiatique), il faudra choisir entre rester sur MSIE 6 ou passer sur un autre navigateur, la principale alternative étant Firefox.

Créer l'objet requêteur

Mais les grands parcs de machines ne basculent pas rapidement, et pour quelques années encore en tout cas, nous allons devoir écrire du code capable d'accommoder tant MSIE avant sa version 7 que les autres navigateurs. Voici, ci-après, une première version, peut-être un peu brutale, d'une fonction portable de création d'un objet `XMLHttpRequest` (sans utiliser Prototype pour l'instant).

Listing 5-2 Une première tentative d'obtention portable

```
function getRequester() {
    try {
        return new ActiveXObject('Msxml2.XMLHTTP');
    } catch (e) {
    }
    try {
        return new ActiveXObject('Microsoft.XMLHTTP');
    } catch (e) {
    }
    try {
        return new XMLHttpRequest();
    } catch (e) {
    }
    return false;
}
```

On voit que l'objet a, au fil du temps, été disponible sous plusieurs noms différents : d'abord `Microsoft.XMLHTTP`, puis plus tard, `Msxml2.XMLHTTP`. Quant à son utilisation comme objet natif JavaScript, elle est des plus simples.

Cet algorithme a l'avantage de se factoriser facilement (le code Prototype qui réalise cette tâche, à savoir `Ajax.getTransport()`, n'est pas sans une certaine esthétique), mais si vous souhaitez minimiser le nombre d'exceptions levées, vous pouvez utiliser une autre variante, très répandue dans les articles et didacticiels sur le sujet.

Listing 5-3 Une version avec les commentaires conditionnels MSIE

```
function getRequester() {
    var result = false;
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
    try {
        result = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
        try {
            result = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (E) {
            result = false;
        }
    }
@end @*/
```

```
    if (!result && 'undefined' != typeof XMLHttpRequest) {
        try {
            result = new XMLHttpRequest();
        } catch (e) {
            result = false;
        }
    }
    return result;
}
```

Notez les syntaxes spécifiques à MSIE, qui constituent des commentaires conditionnels. Ici, MSIE ignorera le commentaire s'il est sur une version inférieure à 5, tandis que les autres navigateurs les ignoreront systématiquement.

Ce code peut sembler futé, mais à mon sens, il s'agit là de beaucoup de bruit pour rien, sans parler de la pollution visuelle engendrée par la syntaxe propriétaire des commentaires conditionnels. Dans une application web, même exigeante, on imagine difficilement plus d'une dizaine de demandes de requêteurs à la seconde (et le plus souvent, bien moins que ça), et le « gaspillage » de temps causé par les levées supplémentaires d'exceptions se chiffre au pire en centièmes de secondes.

C'est pourquoi, à l'issue de cette section consacrée à examiner les entrailles de l'API `XMLHttpRequest`, nous utiliserons les enrobages de confort proposés par Prototype ; même si, par exemple, l'obtention d'un requêteur y utilise un algorithme similaire à notre première tentative, le gain de productivité et l'écart infinitésimal des performances justifient pleinement cette décision.

Décrire notre requête

Techniquement, l'ordre recommandé d'initialisation d'un objet `XMLHttpRequest` suppose une étape supplémentaire avant de préparer la requête ; pourtant, nous allons suivre un ordre plus logique pour l'instant et céder aux exigences techniques un peu plus tard.

Pour configurer une requête, on doit d'abord l'ouvrir (et ce n'est pas tous les jours qu'on vous encourage à l'ouvrir), en indiquant :

1. la méthode HTTP utilisée (par exemple, `GET` ou `POST`) ;
2. l'URL de la ressource dynamique à requêter côté serveur ;
3. le mode de synchronisation (asynchrone par défaut, mais on peut passer en synchrone, bien que ce soit plutôt une hérésie !) ;
4. une authentification HTTP optionnelle (identifiant, mot de passe), si la ressource côté serveur en nécessite une.

Cette initialisation s'effectue avec la méthode `open`. Par exemple :

```
var requester = getRequester();
requester.open("POST", "/blog/comment", true);
```

Une requête HTTP, outre sa méthode et son URL cible, est constituée (comme nombre de requêtes/réponses dans les protocoles qui font vivre le Web) d'en-têtes et d'un corps. Les en-têtes permettent de décrire une foule de choses, comme le type de résultat attendu, les formats de données acceptés, le type d'encodage employé pour le corps de la requête, etc. On les définit à l'aide de la méthode `setRequestHeader`.

Ainsi par exemple, il est parfois nécessaire de circonvenir la politique agressive de cache des résultats de requêtes `GET` mise en œuvre par MSIE, notamment dans le cadre de requêtes internes effectuées à intervalles potentiellement très faibles. On peut réaliser cela en utilisant un en-tête approprié pour la requête, comme ceci :

```
requester.setRequestHeader("If-Modified-Since", "Sat, 1 Jan 2000
00:00:00 GMT");
```

Quant au corps de la requête, il faut d'abord savoir qu'il n'y en a pas forcément. Ainsi, une requête de type `GET` encode tous ses paramètres dans l'URL, ce qui suppose un envoi de corps vide (`null`). En revanche, une requête `POST` ou `PUT` utilise fréquemment des données fournies par le corps de la requête (les champs d'un formulaire y figurent, par exemple).

Envoyer la requête

Le corps de la requête est fourni au moment de **l'envoi** de celle-ci, à l'aide de la méthode `send`. Voici deux exemples de requête, une en `GET` et une en `POST` :

Listing 5-4 Deux exemples de requête, l'une en GET, l'autre en POST

```
requester.open("GET", "/blog/comments?article_id=183");
requester.send(null);
...
requester.open("POST", "/blog/comment");
requester.send("article_id=183&author_name=TDD&author_email=tdd@example
.com&comment=Voici+un+exemple+de+requete+POST");
```

Ne soyez pas alarmés par l'aspect du deuxième appel à `send` : cet encodage particulier des données, qui correspond à l'encodage par défaut pour les méthodes `POST` et `PUT`,

à savoir le format `application/x-www-form-urlencoded`, peut être réalisé automatiquement par JavaScript.

Une fois la requête lancée, il ne reste plus qu'à en suivre la progression pour finalement récupérer les résultats éventuels.

Recevoir et traiter la réponse

C'est ici que la réalité technique nous rattrape, car pour suivre la progression de la requête (de sa création à la fin de la récupération de la réponse du serveur), il faut s'y être pris à l'avance, en créant ce qu'on appelle une fonction de rappel, c'est-à-dire une fonction de votre cru, respectant une certaine forme, qui sera appelée par le système dans certaines circonstances.

Ici, il s'agit d'une fonction sans argument particulier, mais qui devra pouvoir accéder à votre objet requêteur pour interroger son état. On l'associe à l'objet en l'affectant à la propriété `onreadystatechange`. Cette association se fait de préférence avant l'appel à `open` (sauf si on tente de réutiliser le même requêteur par la suite dans MSIE, auquel cas il vaut mieux la faire après).

La fonction va interroger l'état actuel du requêteur, en examinant sa propriété `readyState`. Au fur et à mesure du cycle de vie du requêteur, cette propriété peut prendre les valeurs suivantes :

1. **non initialisé** (pas encore d'appel à `open`) ;
2. **ouvert** (un appel à `open` a été correctement exécuté) ;
3. **envoyé** (la requête a été préparée, l'appel à `send` à eu lieu) ;
4. **en réception** (des données arrivent, mais ce n'est pas fini) ;
5. **réception terminée** (toutes les données sont arrivées, on peut à présent consulter la réponse du serveur).

Voici un code mettant en place une fonction de rappel qui, lorsque la requête aura abouti, affichera le résultat dans une boîte de message :

```
requester.onreadystatechange = function() {
    if (4 == requester.readyState && 200 == requester.status)
        alert(requester.responseText);
};
```

Le résultat de la requête comprend un statut HTTP (le code 200 indique que tout s'est bien passé), consultable via la propriété `status`, et un corps de réponse. La propriété `responseText` fournit le « texte brut » de cette réponse, quel qu'en soit le

format (XML, JSON, XHTML, etc.), mais si la réponse est un format XML et qu'on souhaite le traiter comme tel (comme un document XML manipulable à l'aide des interfaces du DOM), on peut utiliser `responseXML`.

Une utilisation complète de notre petit serveur d'exemple

Nous allons à présent assembler ces fragments de code dans un tout cohérent. Pour commencer, il va nous falloir quelques fichiers statiques : une page HTML mais aussi un fichier JavaScript (et même deux, car nous allons utiliser Prototype pour nous simplifier les parties non Ajax).

Le script serveur que nous avions écrit ne prévoit pas de laisser un accès à un répertoire du disque : seuls deux points d'accès sont prévus, `/hi` et `/time`, qui amènent tout droit sur du code Ruby. Nous devons donc arrêter ce script (un simple Ctrl+C dans la console), le modifier comme ci-après, et le relancer (en tapant simplement `ruby hi_timed.rb`, comme tout à l'heure) après avoir créé, au même niveau, un répertoire `docroot` destiné à accueillir nos fichiers statiques.

Listing 5-5 Script serveur modifié pour gérer un répertoire racine

```
#! /usr/bin/env ruby

require 'webrick'
include WEBrick

server = HTTPServer.new(:Port => 8042)

server.mount('/', HTTPServlet::FileHandler, './docroot') ❶

server.mount_proc('/hi') do |request, response|
  response.body = 'Bonjour !'
end

server.mount_proc('/time') do |request, response|
  response.body = Time.now.to_s
end

trap('INT') { server.shutdown }

server.start
```

La ligne ❶ associe à la racine de nos URL le répertoire `docroot` situé au même niveau que notre script `hi_timed.rb`.

Le nom du répertoire n'a bien sûr rien d'obligatoire, puisque nous le précisons dans le script. Nous allons y déposer trois fichiers :

1. Notre fichier HTML, qui fournit simplement la page de test.
2. Le fichier JavaScript de Prototype.
3. Notre fichier JavaScript, qui va associer des comportements aux boutons de notre page, et effectuer les requêtes Ajax.

Voici le code, très simple, de notre fichier HTML.

Listing 5-6 Notre page hi_timed_client.html pour cet exemple trivial d'Ajax

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
    ➥ charset=iso-8859-15" />
    <title>Exemple Ajax trivial</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="hi_timed_client.js"></script>
</head>
<body>

<h1>Exemple Ajax trivial</h1>

<form>
<p>
    <input type="button" id="btnGreet" value="Salut !" />
    <input type="button" id="btnWhatTime" value="Quelle heure est-il ?" />
</p>
</form>

</body>
</html>
```

Vous pouvez récupérer le fichier JavaScript dans l'archive des codes source de ce livre, sur le site des éditions Eyrolles.

Enfin, voici le code source de notre fichier JavaScript, avec quelques commentaires de rappel sur la partie Ajax. Si vous avez des doutes sur les autres parties (association de gestionnaires d'événements, etc.), feuilletez donc les chapitres 3 et 4 pour retrouver vos marques.

Listing 5-7 Notre fichier JavaScript, hi_timed_client.js

```
function askTime(e) { ❶
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status)
            alert("Maintenant : " + requester.responseText);
    };
    requester.open("GET", "/time", true);
    requester.send(null);
} // askTime

function bindButtons(e) {
    Event.observe($('btnGreet'), "click", greet, false);
    Event.observe($('btnWhatTime'), "click", askTime, false);
} // bindButtons

function getRequester() { ❷
    var result = false;
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
    try {
        result = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
        try {
            result = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (E) {
            result = false;
        }
    }
@end @*/
    if (!result && 'undefined' != typeof XMLHttpRequest) {
        try {
            result = new XMLHttpRequest();
        } catch (e) {
            result = false;
        }
    }
    return result;
} // getRequester

function greet(e) { ❸
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status)
            alert(requester.responseText);
    };
    requester.open("GET", "/hi", true);
    requester.send(null);
} // greet

Event.observe(window, "load", bindButtons, false);
```

❶ Voici la fonction qu'on associera à notre bouton « Quelle heure est-il ? ». On y retrouve les différents blocs de code vus plus haut : l'obtention d'un requêteur, l'association d'un gestionnaire d'état, la détection d'une réponse terminée et valide ainsi que l'envoi de la requête.

❷ Revoici notre fonction d'obtention de requêteur, avec ses commentaires conditionnels MSIE. On pourrait polémiquer longuement sur le meilleur algorithme d'obtention, mais ce serait inutile puisque, à partir du prochain chapitre, nous basculerons sur les fonctions Ajax de Prototype de toutes façons.

❸ Voici le gestionnaire associé au clic sur le bouton « Salut ! ». Il est presque identique à celui de « Quelle heure est-il ? ». La seule différence est qu'il ne préfixe pas le texte de retour.

Voilà, les acteurs sont en place, la première peut commencer : si ce n'est déjà fait, lancez votre script serveur, puis ouvrez un navigateur et allez sur http://localhost:8042/hi_timed_client.html. Vous devriez obtenir un affichage similaire à celui-ci.



Figure 5–1
Notre page de test

Cliquez sur le bouton « Salut ! ». La page ne se recharge absolument pas, mais un petit appel du pied au serveur nous renvoie le texte « Bonjour ! », que notre gestionnaire d'état s'empresse d'afficher avec un `alert`.

Essayez à présent avec le bouton « Quelle heure est-il ? » : vous obtenez un affichage (certes en anglais, et plus précisément au format défini par la RFC 2822) de la date et l'heure courante. Plusieurs tentatives amènent évidemment un résultat différent, car le temps passe...

Comment surveiller les échanges Ajax de nos pages ?

Lorsqu'on met au point une fonctionnalité basée sur Ajax et que des bogues apparaissent, il est vite frustrant de ne pas avoir un œil sur le contenu détaillé des réponses que nous fournit le serveur.

Nous avons déjà évoqué l'extension Firebug et ses inspecteurs et débogueurs pour le DOM, CSS, JavaScript et HTML, tous facilement visibles dans un espace en bas de page. Cette extension permet aussi de surveiller les requêtes et réponses récupérées par `XMLHttpRequest`.

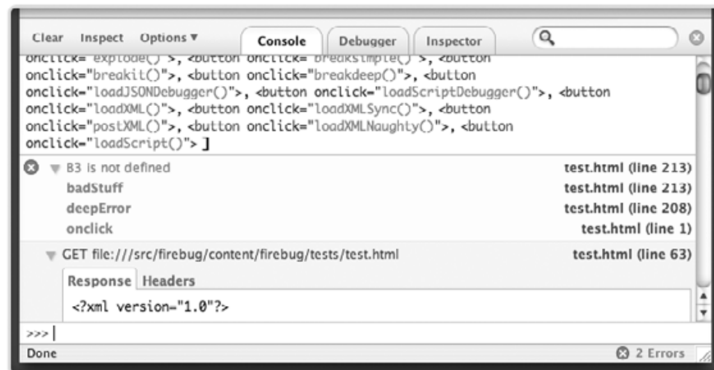


Figure 5–2
Le panneau Firebug en plein travail

Pour activer le suivi des requêtes Ajax (désactivé par défaut), déroulez le menu `Options` du panneau et cochez l'option `Show XMLHttpRequests`. Après quoi, voici l'aspect du panneau après un clic sur chacun de nos deux boutons :

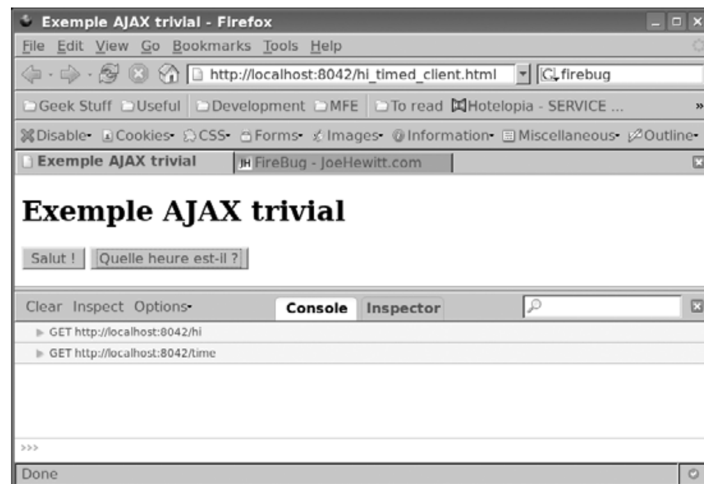


Figure 5–3
Firebug nous signale nos deux requêtes Ajax.

En déroulant ces lignes de résultat et en choisissant l'onglet Headers pour l'un d'eux, on a une idée des informations (détaillées !) qui nous sont proposées :

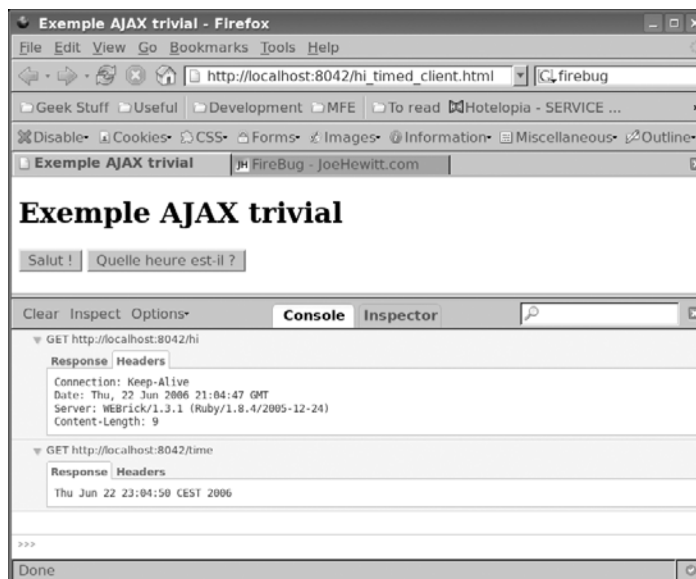


Figure 5–4
Tant les en-têtes que le corps de réponse sont disponibles.

Il existe de nombreux outils d'aide au développement web, et même si à l'heure où j'écris ces lignes, MSIE ne dispose pas, à ma connaissance, d'une aide au suivi des requêtes Ajax, il existera peut-être une solution, même partielle, d'ici la parution de ce livre.

Types de réponse : XHTML, XML, JS, JSON...

Les lecteurs attentifs (je suis sûr que vous en faites partie) auront sans doute remarqué dans l'avant-propos une mention de réponses Ajax utilisant autre chose que du XHTML ou du XML. Après tout, si nous examinons les réponses de notre petit exemple précédent, nous voyons qu'il ne renvoie pas un format particulier : il renvoie une seule donnée, textuelle, très simple. Pas de balisage XML, pas de balises XHTML, pas de code JavaScript...

En effet, dans la mesure où c'est votre code qui interprète la réponse, vous avez le champ libre quant au format de celle-ci. Il vous appartient de choisir au mieux, en utilisant un format qui facilite le traitement côté client, sans pour autant être pénible à générer côté serveur.

Il existe plusieurs types de réponse assez répandus, que nous allons exposer un peu plus loin, avec un exemple concret pour chacun. Mais gardez à l'esprit qu'il n'y a pas vraiment de contraintes, hormis la difficulté que poserait probablement le traitement de données binaires en JavaScript : on se limitera normalement à des réponses de type texte.

Bien choisir son type de réponse

Il s'agit de réfléchir au cas par cas. Dans l'exemple précédent, nous souhaitions simplement récupérer une information unique, pour ne pas dire atomique, sans structure ou complexité aucune. Ne pas s'encombrer d'un format spécifique est alors une bonne solution ! D'autres fois, il faudra choisir judicieusement le format. Comment faire ?

Commençons déjà par nous poser la question de ce que constitue un « bon candidat ». Un bon format pour une réponse Ajax devrait, idéalement, satisfaire aux critères suivants :

- Il est facile à traiter côté client, en JavaScript donc. On verra qu'à l'heure actuelle, cela limite (nativement en tout cas) les réponses XML vers MSIE et Safari. En revanche, du XHTML généré côté serveur peut être facilement injecté dans le DOM de la page, du JavaScript peut être évalué, ainsi que JSON, fatalement (pourquoi fatalement ? Un peu de patience...).
- Il n'est pas inutilement encombrant : plus la représentation est simple, plus son analyse est facile, notamment pour un format structuré qui n'est pas couvert par les objets JavaScript disponibles.
- Il est facile à générer côté serveur. Ce dernier point est généralement garanti : à moins d'utiliser une technologie quelque peu contraignante côté serveur (je ne peux m'empêcher de penser aux pages ASP en Visual Basic), celui-ci dispose de toute la richesse fonctionnelle nécessaire à la création de tous types de contenus.

Sur ces bases, nous pouvons passer en revue les différents formats de réponse courants :

- **Texte simple** : par définition, trivial à générer comme à traiter côté client. Particulièrement approprié pour les données toutes simples : petits morceaux de texte, valeurs numériques (pouvant représenter aussi des dates et heures, des booléens et bien d'autres choses), etc.
- **XHTML** : côté client, c'est trivial. On peut insérer le fragment retourné au beau milieu du DOM de la page, en douceur. Ce n'est pourtant pas toujours approprié, techniquement ou conceptuellement, comme nous le détaillerons plus loin.
- **XML** : relativement facile à générer côté serveur, sa facilité de traitement côté client dépend directement de sa taille et du navigateur. MSIE et Safari ne supportent pas le DOM niveau 3 XPath, ce qui rend tout de suite plus compli-

quée l'analyse de XML complexe. Ce n'est en revanche pas un souci pour des documents de faible envergure. C'est d'ailleurs un des deux formats les plus appropriés pour des données structurées.

- **JavaScript** : assez facile à générer côté serveur, il est absolument trivial à traiter côté client : il suffit de l'évaluer, à l'aide d'un simple appel de méthode. Évidemment, il faut être sûr de la fiabilité du script ainsi récupéré. On limite en général à ses propres serveurs… Parfait pour permettre à la couche serveur de renvoyer des traitements de nature dynamique (on verra des exemples plus loin).
- **JSON** : facile à générer côté serveur et trivial à traiter côté client, puisqu'il s'agit en fait d'une expression JavaScript valide. Idéal pour passer des informations structurées spécifiques, le code de traitement JavaScript pouvant être assez réduit par comparaison à l'analyse d'une grappe XML équivalente.

Mais trêve de conseils généraux, rien ne vaut la pratique ! Nous allons mettre en œuvre un exemple concret pour chaque cas de figure, afin de vous aider à mieux cerner les avantages et inconvénients de chaque format. Ce sera aussi l'occasion de détailler leurs contextes techniques.

Gardez à l'esprit que nombre de ces exemples seraient probablement plus simples en utilisant une bibliothèque comme Prototype. C'est d'ailleurs ce que nous ferons, sur des données plus riches, au chapitre suivant. Mais il est important que vous saisissiez bien, au préalable, les tenants et aboutissants de chaque démarche.

Une réponse textuelle simple : renvoyer une donnée basique

Ce type de réponse est approprié à bien des cas de figure. Afin d'ancrer la notion, nous allons voir deux exemples : le premier utilisant des cookies mais simple côté client, le second très simple côté serveur, mais un peu plus lourd côté client.

Exemple 1 : sauvegarde automatique

Dans une application Ajax, il arrive fréquemment que des requêtes Ajax n'attendent pas de résultat particulier. De telles requêtes servent généralement à « tenir le serveur au courant » de ce qui se passe côté client, par exemple de la personnalisation de l'interface qu'effectue l'utilisateur (comme le changement de la disposition des blocs de contenu), l'ajout de contenu, etc. Lorsque l'information est créée côté client et que le serveur n'a rien à y apporter, il se contente de l'enregistrer.

En revanche, la partie client aimerait sans doute savoir si l'enregistrement s'est bien passé. Peut-être la couche serveur a-t-elle expiré sa session, nécessitant une nouvelle authentification ? Peut-être a-t-elle un souci d'accès à la base de données, ou un problème de quota disque ? Ces problèmes n'engendreront pas forcément une erreur de la couche serveur (que nous pourrions détecter en examinant la propriété `status`).

Afin de construire un système robuste, nous devons développer un code paré à toute éventualité, et donc à même de savoir si le traitement n'a pu s'effectuer.

En termes de requête/réponse, c'est assez simple : il suffit de définir que ces requêtes attendent une réponse exprimant un code de statut. On peut s'inspirer des codes HTTP, et décider par exemple que nous renverrons une ligne de texte démarrant par un code numérique générique (200 = OK, etc.), suivi d'un éventuel message à usage plutôt interne, potentiellement utile au débogage.

Afin d'illustrer un tel comportement, nous allons réaliser un petit système de sauvegarde automatique d'un champ texte. Le scénario d'utilisation de la page est le suivant :

1. On arrive sur la page pour la première fois. Le champ a une valeur par défaut, par exemple « Saisissez votre nom ici ».
2. On modifie le nom et on quitte le champ en cliquant ailleurs sur la page, en cliquant sur un lien, en changeant d'onglet... Tout, sauf fermer directement la page (et encore, une seule ligne de script en plus gérerait ce cas si nous le jugions pertinent).
3. Lorsqu'on revient sur la page, le champ a une valeur à jour.

Pour réaliser cela, trois conditions doivent être satisfaites :

- Le serveur doit conserver une information associée au côté client. On va passer par des cookies pour identifier le client et maintenir un tableau associatif côté serveur.
- Le contenu du champ doit être généré dynamiquement. On pourrait se contenter de demander à la couche serveur sa valeur une fois la page chargée, mais cela entraînerait une latence entre l'affichage de la page et la mise à jour de la valeur du champ, qui serait potentiellement désagréable. Il faut donc que la page entière soit générée.
- Le navigateur doit réagir lorsque le champ de saisie perd le focus (cesse d'être la cible de la saisie clavier) en synchronisant la valeur sur le serveur ; si cette synchronisation échoue, il doit le signaler à l'utilisateur.

Afin de générer du contenu côté client, nous allons stocker le modèle de la page dans un fichier HTML, et utiliser une syntaxe spéciale pour y placer des portions dynamiques. On utilisera la bibliothèque ERb, fournie avec Ruby, dont la syntaxe ressemble beaucoup à ASP ou JSP.

Commencez par arrêter, si ce n'est déjà fait, notre serveur de test précédent. Créez un nouveau répertoire de travail pour cet exemple, appelons-le `sauvegarde_auto`. Nous allons d'abord y déposer notre fichier modèle, `modele.rhtml` (car `.rhtml` est l'extension conventionnelle des fichiers ERb).

Listing 5-8 Notre fichier de modèle pour la page d'exemple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
    ➥ charset=iso-8859-15" />
    <title>Sauvegarde automatique</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Sauvegarde automatique</h1>

<p>La saisie ci-dessous doit persister toute seule, si vous avez
JavaScript et
les cookies activés.</p>

<form>
<p><input type="text" id="edtName" value="<%= name %>" /></p>
</form>

</body>
</html>
```

La balise `form` n'est là que pour respecter la DTD de HTML. Notez l'attribut `value` du champ : `<%= name %>`. Il s'agit d'une syntaxe ERb, que ce dernier remplacera dynamiquement à l'exécution. Avant d'attaquer le code du serveur, on remarque toutefois que notre fichier aura besoin de deux scripts pour dialoguer avec le serveur : Prototype bien sûr, pour simplifier le script comme nous avons pris l'habitude de le faire, et un script dédié à notre exemple.

Créons donc un sous-répertoire `docroot` et plaçons-y `prototype.js`, plus un script `client.js` que nous pouvons dériver de `hi_timed_client.js` (pour récupérer la fonction `getRequester()`). Nous ajusterons ce second script un peu plus tard. Voici notre serveur, stocké dans un fichier `serveur.rb`, à la racine de notre répertoire de test, où se trouve aussi `modele.rhtml` :

Listing 5-9 Notre serveur pour cet exemple

```
#! /usr/bin/env ruby

require 'erb' ❶
require 'webrick'
include WEBrick

DEFAULT_NAME = 'Saisissez votre nom ici'
names = {} ❷
sessionGen = 0
template_text = File.read('modele.rhtml') ❸
page = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/page') do |request, response| ❹
  name = request.cookies.empty? ? DEFAULT_NAME :
         ➥ names[request.cookies[0].value.to_i]
  response['Content-Type'] = 'text/html'
  response.body = page.result(binding)
end

server.mount_proc('/save_name') do |request, response|
  response['Content-Type'] = 'text/plain'
  if 0 == rand(4) ❺
    response.body = '501 Could not be saved.'
    next
  end
  if request.cookies.empty? ❻
    sessionGen += 1
    session_id = sessionGen
    cookie = Cookie.new('session_id', session_id.to_s)
    cookie.expires = Time.utc(2010, 12, 31)
    response.cookies << cookie
  else
    session_id = request.cookies[0].value.to_i
  end
  names[session_id] = request.query['name']
  response.body = '200 Saved.'
end

trap('INT') { server.shutdown }

srand
server.start
```

❶ Nous avons besoin du module ERb pour interpréter la syntaxe de notre modèle HTML.

❷ Conteneur associant clés de sessions et noms sauvegardés, et générateur de clés.

❸ Chargement du texte du modèle depuis le fichier, et construction du moteur pour l'interpréter. L'extension `.rhtml` est juste une convention...

❹ Il faut fournir la variable `name`, qui contient soit la valeur par défaut, soit celle associée à la session. L'appel à `result` construit la page. `binding` représente la portée courante, dans laquelle ERb va chercher la variable `name`.

❺ Simulation d'un échec environ une fois sur quatre.

❻ Si c'est la première sauvegarde (pas de cookie existant), on crée une clé de session qu'on envoie dans un cookie valable jusqu'au 31/12/2010 (ça nous donne le temps de tester...). Ensuite on met à jour l'information transmise dans le conteneur de noms, côté serveur.

Cet exemple ne tiendrait pas une forte charge, dans la mesure où le générateur de clés de sessions n'est pas protégé contre des accès concurrents. Mais pour notre exemple, qu'importe !

Plus important : le choix du stockage de nom côté serveur, plutôt que d'utiliser directement le cookie. C'est une bonne habitude à prendre, principalement pour deux raisons :

1. Un cookie n'a qu'une capacité limitée de stockage, pour des raisons de performances essentiellement. Il est acceptable d'y stocker un numéro (comme ici) ou un texte de faible longueur (clés de session plus communes en ASP, PHP, JSP, Rails...). En revanche, stocker des contenus plus lourds va poser problème. Pour éviter deux poids deux mesures, on stocke donc nos données côté serveur.
2. Un cookie est stocké en clair sur le poste client, ce qui peut constituer un souci si on venait à y stocker des données sensibles voire confidentielles. Certaines attaques XSS (*Cross-Site Scripting*, où un script est injecté sur un site pour transmettre ses cookies à un tiers) permettent à d'autres sites que le nôtre de récupérer des cookies qui ne leur appartiennent pas. Évitons donc d'y stocker des données « utiles ».

À présent que nous avons notre couche serveur au grand complet, nous pouvons la tester manuellement, pour ensuite passer à l'écriture du script qui synchronisera automatiquement côté serveur, en Ajax.

Lancez votre serveur depuis une ligne de commande placée sur le répertoire, en tapant simplement `ruby serveur.rb`.

Dans votre navigateur, allez sur l'URL http://localhost:8042/page.



Figure 5–5
Votre page au premier accès : le texte par défaut

Nous n'avons pas encore écrit le script qui va synchroniser notre saisie, alors réalisons une invocation à la main, en naviguant à présent sur l'URL suivante : http://localhost:8042/save_name?name=Christophe.
Vous devriez obtenir un message de code 200 ou 501. Vous pouvez rafraîchir la page plusieurs fois pour voir s'afficher l'un ou l'autre, suivant le résultat du « lancer de dé » côté serveur. Une fois que vous avez obtenu au moins un code 200, revenez sur votre page principale (http://localhost:8042/page) et, si nécessaire, rafraîchissez-la.



Figure 5–6
L'affichage initial prenant en compte notre sauvegarde

Vous pouvez examiner la valeur de vos cookies pour l'hôte `localhost`, afin de voir ce que le serveur vous a renvoyé lors de la sauvegarde initiale. Le mode d'emploi varie d'un navigateur à l'autre. Sur Firefox avec l'extension Web Developer, il suffit de cliquer Cookies > View cookie information dans la barre associée.

On voit bien notre cookie `session_id`, de valeur `1`. L'information à proprement parler (le nom sauvegardé) n'existe pas dans le cookie : seul le serveur en dispose.

Il nous reste à écrire le script ! On va conserver la fonction `getRequester` de notre script de première démonstration, mais au lieu de ses fonctions `askTime`, `bindButtons` et `greet`, nous allons écrire le code suivant, et ajuster l'appel à `Event.observe` en fin de script.

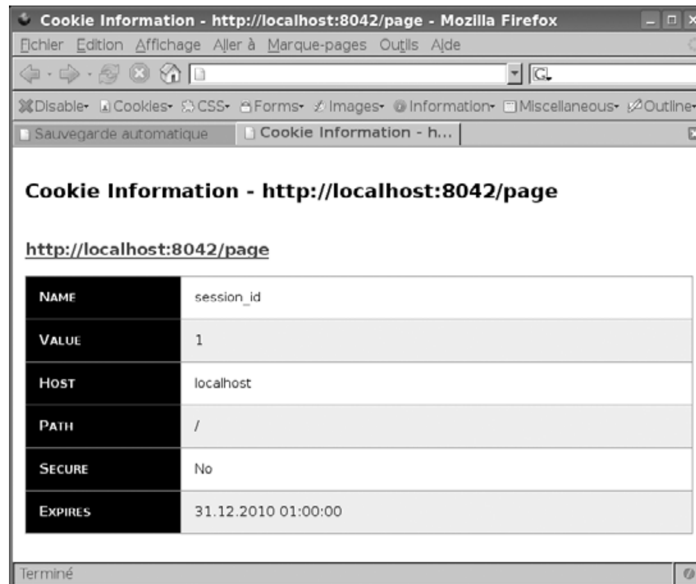


Figure 5–7
L'information stockée dans un cookie sur notre navigateur

Listing 5.10 Notre script dédié, client.js

```
function bindTextField(e) { ❶
    Event.observe($('edtName'), "blur", syncName, false);
} // bindTextField

function getRequester() {
...
} // getRequester

function syncName(e) {
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState
            && (200 != requester.status ❷
                || null == requester.responseText.match(/^200/)))
            alert('Le nouveau nom n\' pas pu être sauvegardé !');
    };
    var qs = $H({ 'name': $F('edtName') }).toQueryString();
    requester.open("GET", "/save_name?' + qs), true);
    requester.send(null);
} // syncName

Event.observe(window, "load", bindTextField, false);
```

❶ On associe notre fonction à l'événement `blur` du champ, c'est-à-dire la perte de focus. Toute saisie suivie d'une perte de curseur (clic en dehors de la zone de saisie, notamment suivi de lien, changement d'onglet, ouverture de menu, etc.) lancera la sauvegarde. En l'associant à l'événement `unload` de `window`, on pourrait aussi sauver sur fermeture directe.

❷ Un problème peut venir de HTTP (on teste donc `status`), mais aussi se situer à l'intérieur de notre couche serveur. En examinant le début du texte renvoyé, on détecte un code à problème.

Notez l'astuce qui consiste à utiliser un `Hash` (fourni par Prototype) pour construire correctement la *query string* encodée de notre URL...

Et voilà, ça marche ! Essayez : rafraîchissez la page, modifiez le nom et cliquez n'importe où ailleurs (ou simplement changez d'onglet, allez sur une autre application, revenez...) et rafraîchissez : votre nouvelle valeur est bien là. Si vous avez installé Firebug, vous pouvez ouvrir son panneau pour suivre les requêtes de la page.

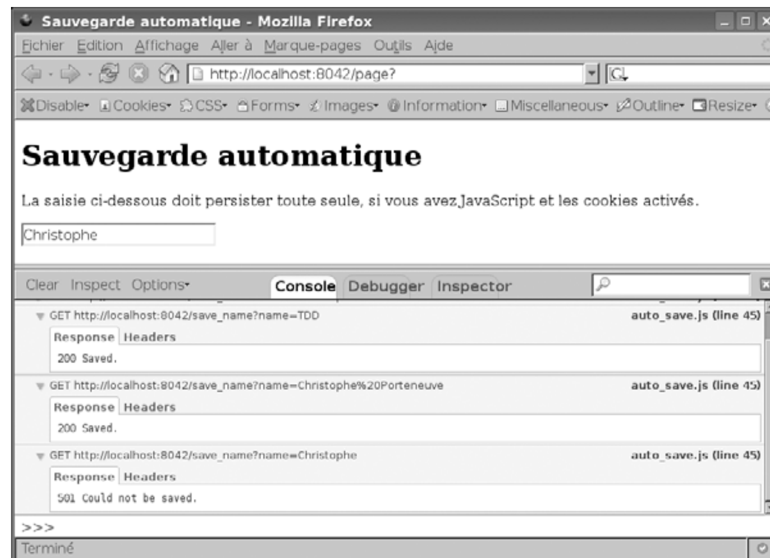


Figure 5–8
Modifications et clics en dehors de la zone de saisie, suivis par Firebug

Précisons pour finir que, dans le cadre d'une véritable application, on utilisera normalement la méthode POST pour des actions modificatives comme celle-ci. Mais cela nous aurait contraint à complexifier un peu notre code de démonstration afin de gérer l'invocation manuelle dans la barre d'adresses et l'invocation Ajax.

J'ai préféré rester simple pour cet exemple. Pour rappel, l'invocation POST aurait ressemblé à ceci côté client :

```
requester.open("POST", "/save_name", true);
requester.send(qs);
```

Exemple 2 : barre de progression d'un traitement serveur

Autorisons-nous à présent un exemple un peu plus « sexy », mêlant Ajax et CSS pour obtenir quelque chose de plus joli qu'une simple sauvegarde automatique (laquelle, il faut bien l'avouer, recèle un potentiel esthétique plutôt léger). Nous allons réaliser une barre de progression dans une page web, qui va se mettre à jour toute seule pour suivre l'avancée d'un traitement sur le serveur. Sans Ajax, ce type de fonctionnalité exige des cadres, voire des `iframes`. Rien de tout cela pour nous : une seule page, sans cadre.

Bien entendu, nul besoin d'effectuer un véritable traitement côté serveur. Au risque de passer pour des fainéants, nous allons nous contenter de simuler un traitement : chaque fois qu'on nous demandera de signaler notre progression, nous nous contenterons d'augmenter un pourcentage, sans rien faire pour autant. Une fois en bout de course, on repart à zéro, histoire de pouvoir rafraîchir la page pour refaire la démonstration. En somme, on prototype, on maquette, mais on ne fait pas vraiment le boulot (c'est déjà mieux que de n'avoir qu'un Powerpoint à montrer).

Créez un nouveau répertoire de travail, avec comme d'habitude un sous-répertoire `docroot` contenant `prototype.js`. Vous pouvez aussi y recopier notre `client.js` récent, afin de récupérer `getRequester`.

Voici notre script `serveur.rb`, regardez comme il est petit !

Listing 5-11 Notre serveur simulant une progression de tâche

```
#! /usr/bin/env ruby

require 'webrick'
include WEBrick

progress = 0

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')
```

```
server.mount_proc('/whatsup') do |request, response|
  response['Content-Type'] = 'text/plain'
  progress += rand(5) + 1
  progress = 100 if progress > 100
  response.body = progress.to_s
  # Boucler pour la prochaine séquence d'appel ;-)
  progress = 0 if 100 == progress
end

trap('INT') { server.shutdown }

srand
server.start
```

Rien de bien extraordinaire, comme vous pouvez le voir. On fait grimper le pourcentage de progression à chaque appel, d'un pas entre 1 et 5. Si on dépasse 100, on rabote la valeur. Et une fois arrivé au bout, on repart à zéro pour la démonstration suivante (voilà un code bien loin d'une implémentation réelle...).

Côté client, il va nous falloir une page web toute simple, sans partie dynamique, contenant notre barre de progression. Nous allons réaliser celle-ci entièrement en CSS, et histoire de sacrifier aux traditions (certes toutes fraîches), nous allons y coller l'incontournable *spinner*, cette petite animation omniprésente sur les sites Web 2.0, qui indique qu'on attend après Ajax. Notez que si vous optez ici pour la version   la plus classique, vous êtes libre d'utiliser l'animation que vous voulez. Sans doute trouverez-vous votre bonheur sur cette petite collection en ligne :
http://www.napyfab.com/ajax-indicators/.

Voici donc notre page. Comme elle est servie statiquement par notre serveur, nous allons simplifier l'URL en la nommant `index.html`, dans `docroot`.

Listing 5-12 La page web client et sa barre de progression CSS

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
    ➥ charset=iso-8859-15" />
    <title>Barre de progression</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
```

```
<body>

<h1>Barre de progression</h1>

<p>Suivez le déroulement du processus côté serveur avec la barre ci-
dessous.</p>

<div class="progressBar" id="progress">
    <span class="pbFrame">
        <span class="pbColorFill"></span>
        <span class="pbPercentage">0%</span>
    </span>
    <span class="pbStatus"></span>
</div>

</body>
</html>
```

Nous partons du principe qu'une barre de progression est représentée par :

1. Un `div` jouant le rôle de conteneur pour la barre et son animation/image de droite (*spinner* ou icône de complétion).
2. Dans ce `div`, un premier `span` qui représente la barre à proprement parler, et un second qui fournira l'animation ou image.
3. Dans le `span` de la barre, un `span` gérant la barre colorée qui va progresser en largeur, et un autre fournissant la représentation textuelle de la progression. En termes d'accessibilité, il est en effet bon d'avoir une représentation textuelle facilement ostensible dans le flux du document.

Cette façon de faire n'est pas forcément la meilleure, mais elle permet de définir de multiples barres dans une même page, en fournissant simplement un `id` distinct pour chaque `div`, et en respectant juste les classes CSS indiquées à l'intérieur.

Listing 5-13 La feuille de styles pour ces barres de progression

```
div.progressBar {
    font-family: sans-serif;
    position: relative;
    margin: 1em 0; width: 204px; height: 20px;
}

span.pbFrame {
    position: absolute;
    left: 0; top: 0; width: 180px; height: 16px;
    border: 2px solid #444;
    background-color: silver;
}
```

```
span.pbColorFill {
    position: absolute;
    left: 0; top: 0; height: 100%; width: 0;
    z-index: 1;
    background-color: green;
}

span.pbPercentage {
    position: absolute;
    left: 0; top: 0; height: 100%; width: 100%;
    z-index: 2;
    font-size: 1em;
    line-height: 16px;
    font-weight: bold;
    text-align: center;
}

span.pbPercentage.over50 {
    color: white;
}

span.pbStatus {
    position: absolute;
    background-repeat: no-repeat;
    top: 2px; width: 16px; height: 100%;
    z-index: 2;
}

span.pbStatus.working {
    background-image: url(spinner.gif);
}

span.pbStatus.done {
    background-image: url(ok.png);
}
```

Si certains aspects vous échappent, allez faire un tour à l'annexe B. Elle vous rappellera entre autres les concepts qui sous-tendent le positionnement et le modèle de boîtes, éclairant l'utilisation faite ici de `position` et `z-index`.

Notez l'emploi des classes multiples qui aide à bien séparer le comportement (changement d'état dans le temps) de l'aspect (images et couleurs différentes). En effet, dans le script ci-après, on ne touche pas directement au style, on se contente de manipuler les classes assignées aux éléments.

Enfin, il reste notre script client.

Listing 5-14 Le script client, qui dès le chargement interroge 10 fois par seconde

```
function getRequester() {
...
} // getRequester
INTERVAL = 100; ❶
var gProgressTimer = 0;

function checkProgress(id) {
    var node = $(id);
    var filler = node.getElementsByClassName('pbColorFill').first(); ❷
    var percent = node.getElementsByClassName('pbPercentage').first();
    var status = node.getElementsByClassName('pbStatus').first();
    var firstHit = 0 == gProgressTimer; ❸
    if (!firstHit) {
        window.clearTimeout(gProgressTimer);
        gProgressTimer = 0;
    } else {
        Element.removeClassName(percent, 'over50');
        Element.removeClassName(status, 'done');
        Element.addClassName(status, 'working');
    }
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            var progress = parseInt(requester.responseText, 10); ❹
            if (100 <= progress)
                progress = 100;
            filler.style.width = progress + '%'; ❺
            percent.firstChild.nodeValue = progress + '%';
            // Blanc sur vert, c'est plus joli... :-)
            if (progress > 50 &&
                ➥ !Element.hasClassName(percent, 'over50'))
                Element.addClassName(percent, 'over50');
            if (100 == progress) { ❻
                Element.removeClassName(status, 'working');
                Element.addClassName(status, 'done');
            } else
                gProgressTimer = window.setTimeout(
                           ➥ 'checkProgress("' + id + '")', INTERVAL);
        }
    };
    requester.open('GET', '/whatsup', true);
    requester.send(null);
} // checkProgress

Event.observe(window, 'load',
              ➥ function() { checkProgress('progress') }, false);
```

❶ Intervalle entre requêtes de progression : 100 ms.

❷ Obtention des nœuds du DOM qui nous intéressent d'après leur classe, à l'intérieur du `div` identifié par `id`.

❸ Distinction entre le premier appel (juste après le chargement de la page) et les suivants, histoire de pouvoir réutiliser la barre sans recharger la page, qui sait ?

❹ On se protège contre « 08 », « 09 », etc. et... contre un serveur mal fichu !

❺ En définissant une largeur en %, on s'adapte automatiquement à la taille du `span` conteneur.

❻ Si on a fini, exit le *spinner*, voici le sceau de bonne fin ! Sinon, n'oublions pas de redemander une requête pour bientôt.

> **FICHIERS D'EXEMPLE Les images**
>
> Sur cet exemple, des fichiers vous manquent : les images `spinner.gif` et `ok.png`. Vous pouvez vous les procurer dans l'archive contenant tout le code de cet ouvrage, disponible en ligne sur le site des éditions Eyrolles.

Il ne vous reste plus qu'à lancer votre serveur (avez-vous pensé à arrêter le précédent ?), et à naviguer sur http://localhost:8042/. Et là, joie !



Figure 5–9
Quelques étapes de la progression, jusqu'à la fin

Voilà qui a tout de même une autre allure qu'un simple champ de formulaire !

Fragments de page prêts à l'emploi : réponse XHTML

Un mécanisme très fréquemment rencontré consiste à produire un fragment XHTML côté serveur, et le renvoyer au client. Ce dernier va « incruster » le fragment à l'endroit approprié.

Bien sûr, cela implique une légère intrusion de la couche présentation dans la couche métier, mais avec les bonnes méthodologies, ce n'est pas très grave. Après tout, la couche présentation est générée côté serveur pour des pages dynamiques. Il suffit de ne pas mettre de XHTML en dur dans le code pour rester plutôt propre, et justement, nous avons déjà vu ERb pour réaliser cela dans notre exemple !

Créez un nouveau répertoire, et comme d'habitude, copiez-y `docroot`, `prototype.js` et `client.js`, que vous réduirez à son `getRequester`. Nous allons aussi repartir de notre `index.html` précédent pour le squelette de la page principale.

Notre exemple sera un formulaire de commentaire, par exemple dans le cadre d'un blog. Le commentaire sera envoyé en Ajax, et cette fois-ci, nous ferons l'effort côté serveur permettant un envoi en `POST`, ce qui est tout de même plus conforme aux règles du W3C pour les actions censées modifier la couche serveur.

Le serveur générera le bloc XHTML de représentation du commentaire (prétendument sauvé), et le renverra pour insertion. Nous utiliserons Prototype pour réaliser l'insertion en question.

Voici d'abord notre code serveur.

Listing 5.15 Notre script serveur, capable de traiter le POST comme le GET

```
#! /usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

template_text = File.read('commentaire.rhtml')
comment = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')
server.mount_proc('/add_comment') do |request, response|
  post = 'POST' == request.meta_vars['REQUEST_METHOD'] ❶
  params = post ? CGI::parse(request.body) : request.query
  params.each { |k, v| params[k] = v[0] } if post
```

```
  # PASSEZ les paramètres, sous peine d'erreur de nil
  name = CGI::escapeHTML(params['name'])
  email = CGI::escapeHTML(params['email'])
  # Formatage simple : les textes séparés par des lignes vierges
  # sont des paragraphes et on supprime les paragraphes vides.
  text = CGI::escapeHTML(params['comment'])
  text.gsub!(/(\r?\n){2,}/, "</p>\n<p>")
  text = ('<p>' + text + '</p>').gsub(/<p>\s*<\/p>/, '')
  response['Content-Type'] = 'text/html'
  response.body = comment.result(binding)
end

trap('INT') { server.shutdown }

server.start
```

Les 3 lignes de code en ❶ permettent de gérer tant le `POST` que le `GET`. Dans la mesure où vous n'utiliserez normalement jamais WEBrick directement en production (préférez, largement, Ruby on Rails !), on n'entrera pas dans les détails. Les habitués des scripts côté serveur de type CGI s'y retrouveront...

Le modèle de commentaire est très simple.

Listing 5-16 Le modèle de commentaire

```
<div class="comment">
    <h3><a href="mailto:<%= email %>"><%= name %></a>
    ➥ @ <%= Time.now.strftime('%H:%M') %></h3>
        <div class="commentText">
            <%= text %>
        </div>
</div>
```

Voici à présent notre page HTML cliente.

Listing 5-17 La page cliente, avec un formulaire plutôt sémantique

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Commentaires</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
```

```
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Commentaires</h1>

<p>Ajoutez un commentaire ci-dessous. En le soumettant, il sera envoyé en
arrière-plan au serveur, et sa représentation XHTML sera renvoyée puis
insérée dynamiquement dans cette page.</p>

<form id="commentForm" method="post" action="/add_comment">
    <fieldset>
        <legend>Un commentaire ?</legend>
        <p>
            <label for="edtName" accesskey="N">Nom</label>
            <input type="text" id="edtName" name="name" tabindex="1" />
        </p>
        <p>
            <label for="edtEmail" accesskey="C">Courriel</label>
            <input type="text" id="edtEmail" name="email" tabindex="2" />
        </p>
        <p class="comment">
            <label for="memComment" accesskey="O">Commentaire</label>
            <textarea id="memComment" name="comment" cols="40"
            ➥ rows="5" tabindex="3">Votre commentaire ici</textarea>
        </p>
        <p>
            <input type="submit" value="Envoyer !" tabindex="4" />
        </p>
    </fieldset>
</form>

<div id="comments"></div>

</body>
</html>
```

Notez que dans une application professionnelle, on fournirait des mécanismes permettant de traiter le formulaire normalement côté serveur, plutôt qu'en Ajax uniquement, à des fins d'accessibilité et de compatibilité maximales : c'est la raison pour laquelle notre formulaire est correctement paramétré.

Le `div` final servira de conteneur pour les insertions dynamiques des blocs XHTML des commentaires.

À présent, voici le script client.

Listing 5-18 Le script client, qui construit la requête POST et insère le résultat

```
function bindForm() {
    Event.observe('commentForm', 'submit', switchToAJAX);
} // bindForm

function getRequester() {
...
} // getRequester

function switchToAJAX(e) {
    Event.stop(e);
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            new Insertion.Bottom('comments', requester.responseText); ❶
        }
    };
    requester.open('POST', '/add_comment', true);
    var data = $H({ ❷
        'name': $F('edtName'),
        'email': $F('edtEmail'),
        'comment': $F('memComment')
    }).toQueryString();
    requester.send(data);
} // switchToAJAX

Event.observe(window, 'load', bindForm);
```

L'appel Prototype en ❶ ajoute le fragment XHTML fourni dans la réponse dans l'élément `comments`, après son contenu existant. Par ailleurs, en `POST`, les paramètres sont par défaut encodés comme dans une URL, mais sont dans le corps de la requête. Passer par un `Hash` comme en ❷ est décidément bien pratique...

Pour que ce soit joli, reste une petite CSS (il ne s'agit véritablement que d'esthétique, le script laissant les styles tranquilles).

Listing 5-19 La feuille de styles employée

```
form#commentForm fieldset {
    background: #ccc;
    width: 60ex;
    border: 1px solid #444;
}

form#commentForm p {
    position: relative;
    height: 2.2em;
}

form#commentForm input#edtName, form#commentForm input#edtEmail,
form#commentForm textarea#memComment {
    font-family: serif;
    position: absolute;
    left: 14ex; right: 0; top: 0; bottom: 0.5em;
}

form#commentForm p.comment {
    height: 6.2em;
}

div.comment {
    font-family: sans-serif;
    width: 60ex; padding: 0 1ex; margin: 1em 0;
    border: 1px solid #990;
    background: #ffc;
}

div.comment h3 {
    font-size: 100%;
}

div.comment h3 a {
    color: maroon;
}

div.comment div.commentText {
    border-left: 0.5ex solid gray;
    margin-left: 1ex; padding-left: 1ex;
    font-style: italic;
}
```

Et voici un exemple après avoir saisi trois séries de valeurs et pressé le bouton d'envoi à chaque fois ; la page ne s'est jamais rechargée !



Figure 5–10
La page après trois saisies envoyées

Un aperçu des coulisses de l'insertion...

Techniquement, Prototype a recours à plusieurs méthodes pour assurer une insertion multinavigateur. Il peut utiliser le `insertAdjacentHTML` spécifique à MSIE, la propriété `innerHTML`, originellement pure MSIE mais aujourd'hui présente dans plusieurs navigateurs (dont Mozilla/Firefox et Opera), ou analyser le fragment pour réaliser une insertion 100 % DOM. C'est un algorithme de travail fastidieux, et on remercie Prototype de

nous éviter son écriture ! D'autant que sur MSIE, `innerHTML` ne fonctionne pas lorsque le fragment qu'on lui donne comporte des balises de formulaire (mais des balises de script, aucun problème ! Toujours logiques, nos amis de Redmond...).

Notez au passage que, lorsqu'elle est disponible, `innerHTML` est bien plus rapide à utiliser qu'une construction DOM brute (de 3 à 30 fois plus rapide, suivant le navigateur !). En revanche, hormis dans Firefox 1.5, elle ne fonctionne pas dans une page servie avec un type MIME XHTML (`application/xhtml+xml`), types qu'exigent normalement les pages XHTML 1.1 et ultérieures. C'est une des nombreuses raisons qui jouent contre la migration massive vers ce type MIME, pourtant sémantiquement plus adapté que `text/html`.

Pour finir avec XHTML, sachez que la notion globale de communication asynchrone HTTP renvoyant un fragment de page est si répandue qu'un microformat lui est dédié : AHAH. Les microformats visent à fournir une ou plusieurs fonctionnalités en se basant sur des formats et standards existants, utilisés de façon innovante et formalisée. Vous en saurez plus en consultant le site dédié aux microformats : http://www.microformats.org/about/, et la page spécifique au microformat AHAH : http://www.microformats.org/wiki/rest/ahah.

Dans la cour des grands : XPath pour traiter des données XML complexes

Tout cela est bel et bon mais on peut se demander pourquoi on ne renvoie pas de XML. Dans Ajax, le X représente XML, et renvoyer une grappe XML peut impressionner. Après tout, tout est à base de XML aujourd'hui : enlevez XML et J2EE n'existe plus, tout comme SOAP et OpenDocument.

Certes. Mais le plus grand danger de XML résulte justement de son immense popularité : on veut en mettre partout. C'est un peu comme Ajax aujourd'hui. Et bien que la pulsion du « tout XML » soit en baisse (heureusement, depuis 1998, les passions se sont calmées), on en trouvera toujours pour miser dessus sans rime ni raison. Le XML n'est pourtant pas incontournable : divers frameworks fonctionnent très bien sans lui.

Ceci dit, je n'écrirais pas une section ainsi nommée si je voulais envoyer XML aux orties ; il existe certains cas de figure où XML est approprié. Il s'agit principalement de cas où les données à récupérer sont particulièrement riches, ou présentent certaines caractéristiques de complexité (par exemple, un nombre inconnu d'éléments, ou des éléments aux noms inconnus à l'avance). Même s'il reste possible de traiter de tels cas autrement, avec JSON par exemple, XML va briller si on doit « fouiller » dans les données de façon un tant soit peu puissante.

En effet, dès lors qu'on dispose d'une grappe XML, on peut utiliser des technologies comme XPath ou encore XSLT pour réaliser des recherches complexes ou des transformations puissantes sur cette grappe. Pour de tels traitements, JSON ne peut tout simplement pas concurrencer XML.

Cette perspective est tout à fait alléchante, mais il convient de mettre un bémol, comme souvent dans l'univers des technologies web côté client : celui de la compatibilité des navigateurs.

Le W3C a défini les spécifications XSLT et XPath ainsi que leurs cousines liées au DOM (par exemple DOM niveau 3 XPath), qui définissent des interfaces telles que `XSLTProcessor`. Ces interfaces sont censées, à terme, être proposées par des objets accessibles en JavaScript, par exemple `document`. Cependant, le support des ces standards varie grandement d'un navigateur à l'autre.

Dans les exemples qui vont suivre, nous allons nous concentrer sur XPath, pour voir comment extraire quelques informations d'une grappe XML sans la parcourir manuellement via les possibilités classiques du DOM (mais si, vous savez bien : `firstChild`, `nextSibling`, `nodeType`, `nodeValue`, et les autres...). Il nous faut donc un navigateur prenant en charge XPath (presque tous, en interne), mais aussi et surtout le DOM niveau 3 XPath.

C'est là que les choses se gâtent : si Firefox (et donc Camino) et Opera (à partir de la version 9) répondent présents, on tombe actuellement à plat sur MSIE, Safari et Konqueror. Il est certes permis d'espérer du changement côté Safari et Konqueror (qui se talonnent toujours de près, n'étant pas sans rapport), mais on sait que côté MSIE, il faudra attendre au grand minimum la version 8, puisque la 7, encore une fois, ne prévoit aucune amélioration significative sur JavaScript ou le DOM.

Utiliser une technologie indisponible sur MSIE peut en refroidir plus d'un, en particulier pour un service Extranet ou Internet. Mais rassurez-vous : des palliatifs existent, sous forme de bibliothèques JavaScript simulant la fonctionnalité. Nous verrons donc d'abord un exemple avec du code « natif » (support DOM niveau 3 XPath), qu'il vous faudra nécessairement tester sur Firefox, Camino ou Opera 9. Par la suite, nous adapterons sa couche client pour réaliser l'équivalent au moyen d'une bibliothèque tierce partie, en l'occurrence GoogleAJAXSLT (voilà un nom qui noue la langue).

Vite et bien : utilisation de DOM niveau 3 XPath

Nous allons implémenter une petite fonction toute bête : une liste de flux Atom pour quelques blogs reconnus, dont nous irons chercher dynamiquement le nombre d'articles. Théoriquement, cela pourrait se passer intégralement côté client, mais cela implique une configuration de sécurité particulière, qui varie sensiblement d'un navigateur à l'autre.

Pour laisser ces problèmes de côté pour l'instant (nous aurons tout le temps de les examiner aux chapitres 8 et 9), nous allons implémenter la récupération des contenus en passant par notre couche serveur, qui jouera le rôle de l'intermédiaire.

Créez un nouveau répertoire et son sous-répertoire `docroot`, copiez-y `prototype.js` et `client.js`, dont on récupérera le `getRequester`, ainsi que le `index.html` pour son squelette.

Voici notre couche serveur, qui a en outre l'avantage de forcer un type de réponse `text/xml`, afin de garantir la construction d'un DOM côté client dans la propriété `responseXML` du requêteur :

Listing 5-20 Notre serveur, simple « proxy » de chargement des flux

```
#! /usr/bin/env ruby

require 'net/http'
require 'uri'
require 'webrick'
include WEBrick

FEEDS = {
    'Standblog' => 'http://standblog.org/dotclear/atom.php',
    'Formats Ouverts' => 'http://formats-ouverts.org/atom.php',
    'IEBlog' => 'http://blogs.msdn.com/ie/atom.xml'
}

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/get_feed') do |request, response|
  # Critique pour pouvoir utiliser requester.responseXML !
  response['Content-Type'] = 'text/xml'
  feed = FEEDS[request.query['feed']]
  response.body = Net::HTTP.get(URI.parse(feed))
end

trap('INT') { server.shutdown }

server.start
```

Attention ! Si vous êtes derrière un proxy, il vous faut ajuster ce code, en remplaçant :

```
Net::HTTP.get
```

par :

```
Net::HTTP.Proxy('votre_hote_proxy', votre_port_proxy).get
```

Rien de bien compliqué... Si le `URI.parse` vous intrigue, sachez simplement qu'une URL est un type particulier d'URI.

Voyons à présent notre page HTML, très simple (pour simplifier le code de cet exemple, on n'a pas rendu la liste des flux dynamique).

Listing 5-21 La page cliente, avec le formulaire de choix de flux

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR" xml:lang="fr-
FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Dernières nouvelles (XML/XPath)</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Dernières nouvelles (XML/XPath)</h1>

<form id="feedForm">
    <p>
        <label for="cbxFeed" accesskey="F">Flux Atom</label>
        <select id="cbxFeed">
            <option>Standblog</option>
            <option>Formats Ouverts</option>
            <option>IEBlog</option>
        </select>
        <input type="button" id="btnProcessFeed"
               ➥ value="Articles récents ?"
         accesskey="A" />
    </p>
    <p id="status"> </p>
</form>

</body>
</html>
```

Il nous faut aussi un brin de CSS pour afficher le traitement en cours et son résultat.

Listing 5-22 La petite feuille de styles

```
p#status {
    height: 16px;
    font-family: sans-serif;
    color: gray;
    background-repeat: no-repeat;
}

p#status.working {
    background-image: url(spinner.gif);
}
```

Un flux Atom est un document XML avec un élément racine `feed` contenant, entre autres, un élément `entry` par article. Il ne s'agit pas forcément de tous les articles du blog, juste des derniers : la taille peut être configurée dans le logiciel de blog.

Voici notre code JavaScript.

Listing 5-23 Le code client JavaScript, utilisant XPath

```
function bindButton() {
    Event.observe($('btnProcessFeed'), 'click', processFeed, false);
} // bindButton

function getRequester() {
...
} // getRequester

function processFeed() {
    var feed = $F('cbxFeed');
    var form = $('feedForm');
    var status = $('status');
    Form.disable(form);
    status.firstChild.nodeValue = '';
    Element.addClassName(status, 'working');
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            var data = requester.responseXML;
            var articleCount = data.evaluate(
                ➥ 'count(//*[name()="entry"])', data, null,
                ➥ XPathResult.NUMBER_TYPE, null).numberValue;
```

```
            Element.removeClassName(status, 'working');
            Form.enable(form);
            status.firstChild.nodeValue = articleCount + ' article(s).';
        }
    };
    var qs = $H({ 'feed': feed }).toQueryString();
    requester.open('GET', '/get_feed?' + qs, true);
    requester.send(null);
} // processFeed

Event.observe(window, 'load', bindButton, false);
```

Remarquez les petites précautions que nous prenons pour empêcher le déclenchement parallèle de plusieurs requêtes : nous désactivons le formulaire le temps du traitement et affichons bien sûr le *spinner* pour patienter.

Par ailleurs, il convient de faire la lumière sur l'invocation quelque peu obscure de la méthode `evaluate` :

- Il s'agit d'une méthode DOM niveau 3 XPath, implémentée par tout élément qui implémente également l'interface DOM `document`.
- Elle prend 5 paramètres obligatoires :
  1. L'expression XPath.
  2. Le nœud servant de contexte à cette expression (souvent le document sur lequel on appelle `evaluate`, en particulier pour une expression démarrant par `/`).
  3. Un solveur éventuel d'espaces de noms (inutile ici, donc `null`).
  4. Le type de résultat souhaité. Parmi les nombreux types, nous choisissons `NUMBER_TYPE`, déclaré dans la classe `XPathResult`, ce qui convient au résultat d'une fonction `count`.
  5. Un objet résultat existant qui sera alors recyclé. Nous n'en avons pas sous la main (ce serait différent si nous bouclions autour d'`evaluate`, par exemple), aussi nous fournissons `null`.
- Elle renvoie un objet `XPathResult` configuré d'après le 4e argument. Dans le cas d'un résultat numérique, on exploite la propriété `numberValue`.

Enfin, l'expression employée dans la fonction `count` signifie : « à tout niveau de profondeur du document (`//`), les éléments dont le nom est `entry` ». En XPath, on peut exprimer cela bien plus simplement, mais Firefox semble avoir un souci, très surprenant d'ailleurs, avec « `//entry` »...

Et voilà ! Une analyse qui nous aurait pris quelques lignes de JavaScript (beaucoup même, si nous n'avions pas Prototype) est ici très courte. Et encore, il ne s'agit que d'une analyse simple. On pourrait imaginer ne sortir que les articles dont le titre comporte un certain texte et mis à jour dans la semaine écoulée, en augmentant à peine l'expression XPath...

En simulant : utilisation de GoogleAJAXSLT

On l'a vu, cette fonctionnalité n'est pour l'instant disponible, en natif, que sur Firefox, Camino et Opera 9 (et ultérieurs, évidemment). Ce qui laisse tout de même de côté Safari, principal navigateur pour les utilisateurs de Mac OS X, Konqueror, très prisé des aficionados de KDE, et surtout MSIE, qui couvre tout de même encore plus de 70 % du marché, monopole oblige.

Pour disposer des mêmes fonctionnalités sur ces navigateurs, il faut pour l'instant avoir recours à des bibliothèques JavaScript tierces. Elles sont légion, mais l'une des plus prometteuses (et dont les performances sont raisonnables, même sur les quelque 21 Ko du flux Atom de l'IEBlog !) est GoogleAJAXSLT (prononcez « Google AJAX S-L-T », ou à l'anglaise : « ... S-L-*Ti* »).

Cette bibliothèque, écrite par le *googlien* Steffen Meschkat, fournit un équivalent pas tout à fait complet, mais largement suffisant, des spécifications XPath et XSLT, directement en JavaScript. Aucune dépendance externe n'est requise (par exemple, la possibilité de recourir aux ActiveX sous MSIE, ou même simplement présence de la bibliothèque Prototype), si ce n'est la prise en charge par le navigateur de l'essentiel de DOM niveau 2, ce qui nous assure un bon fonctionnement sur la quasi-totalité des navigateurs récents.

La bibliothèque se trouve à l'adresse : http://code.google.com/p/ajaxslt/, et est aussi présente dans l'archive des codes source de ce livre. Assurez-vous d'utiliser au moins la version 0.5, sortie le 13 septembre 2006, et qui corrige quelques bogues sévères, lesquels gâcheraient nos exemples. Il s'agit d'un projet libre, sous licence BSD. Déposez les fichiers de script mentionnés ci-dessous dans un sous-répertoire `ajaxslt` de votre `docroot`.

Son utilisation diffère finalement assez peu du code que nous avons écrit pour l'exemple natif. Bien entendu, le serveur ne change pas, tout comme la page cliente (aux scripts près, comme indiqué dans un instant) et la feuille de styles. Il faut en revanche charger des scripts supplémentaires, fournis par la bibliothèque. Il est préférable de les charger dans l'ordre suivant : `util`, `xmltoken`, `dom` et `xpath`.

L'en-tête de notre `index.html` prend donc la forme suivante.

Listing 5-24 Nouvel en-tête pour notre page cliente

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
        ➥ charset=iso-8859-15" />
    <title>Dernières nouvelles (XML/XPath avec GoogleAJAXSLT)</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="ajaxslt/misc.js"></script>
    <script type="text/javascript" src="ajaxslt/xmltoken.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt/xpath.js"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
```

Ceci fait, il nous reste à ajuster le code client. Les modifications sont chirurgicales :

1. Désactivez la journalisation du moteur (affichage dans un pavé à position fixe des grandes étapes du traitement XPath), à l'aide d'une variable globale.
2. Remplacez l'appel à `document.evaluate` par la construction d'un contexte pour le XML et un appel à `xpathParse`.
3. Remplacez la propriété `numberValue` par la méthode `numberValue()`.

C'est tout ! Voici l'extrait modifié du code client, avec les altérations mises en évidence (notez que la variable se termine par un double souligné).

Listing 5-25 La portion modifiée de client.js

```
if (4 == requester.readyState && 200 == requester.status) {
    var data = requester.responseXML;
    logging__ = false;
    var ctx = new ExprContext(data);
    var articleCount = xpathParse(
                      ➥ 'count(//entry)').evaluate(ctx).numberValue();
    Element.removeClassName(status, 'working');
    Form.enable(form);
    status.firstChild.nodeValue = articleCount + ' article(s).';
}
```

Et voilà, tout se passe comme avant, mais cette fois-ci cela fonctionne aussi sur MSIE, Safari, Konqueror...

Piloter la page en renvoyant du JavaScript

Générer du JavaScript côté serveur ! Aurais-je perdu l'esprit ? Je vous imagine aisément froncer les sourcils d'un air réprobateur devant une telle méprise apparente des règles du jeu.

Et pourtant, dans une application complexe, les comportements à adopter dans la page client au retour d'une requête Ajax peuvent être non seulement complexes, mais aussi varier d'une requête à l'autre. Dans tel cas, il faudra ajouter un fragment XHTML, dans tel autre lancer un requêteur Ajax périodique, masquer le message de notification et modifier un texte, etc.

Le fait est que traiter tous ces cas de figure dans le JavaScript de la fonction de rappel (ou dans des fonctions séparées que cette dernière appellerait) conduit à un code JavaScript volumineux côté client. La factorisation est potentiellement difficile, dans la mesure où on souhaite avoir des fichiers JS séparés de nos pages afin de favoriser le travail des caches. Bref, c'est vite une galère !

En générant le JavaScript côté serveur (quelle que soit la technologie employée), on a beaucoup plus de flexibilité : on peut renvoyer le bout de script exactement adapté à la situation. Ce script peut être autonome (à Prototype près, le plus souvent), ou reposer sur des variables et fonctions dont il connaît la disponibilité côté client.

La génération du morceau de script peut varier en difficulté : ainsi, dans l'exemple ci-après, nous allons le générer à la main, ce qui n'est ni très élégant ni très facile. Dans certains frameworks, cette fonction est déjà là. Ruby on Rails, notamment, a introduit cette année les *templates RJS*, qui permettent d'écrire en quelques lignes de Ruby un modèle de réponse Ajax qui sera traduit à la volée en JavaScript portable basé sur Prototype ! Bonheur !

Pour cet exemple, nous allons reprendre notre barre de progression et l'adapter pour faire en sorte que le code côté client n'ait plus de logique algorithmique pour le traitement du pourcentage : c'est le serveur qui déterminera le code à exécuter suite à l'envoi de la requête Ajax. Le côté client se contentera de mettre à disposition les objets nécessaires (composants de la barre de progression concernée), et d'évaluer le code JavaScript renvoyé par le serveur.

Recopiez complètement votre répertoire de travail pour l'exemple de la barre de progression. Nous n'allons en changer que deux fichiers : le serveur et le script.

Commençons par le script : il s'agit simplement de remplacer, dans le traitement du retour de requête Ajax, l'ancien code suivant :

```
var progress = parseInt(requester.responseText, 10);
if (100 <= progress)
    progress = 100;
filler.style.width = progress + '%';
percent.firstChild.nodeValue = progress + '%';
// Blanc sur vert, c'est plus joli... :-)
if (progress > 50 && !Element.hasClassName(percent, 'over50'))
    Element.addClassName(percent, 'over50');
if (100 == progress) {
    Element.removeClassName(status, 'working');
    Element.addClassName(status, 'done');
} else
    gProgressTimer = window.setTimeout(
                    ➥ 'checkProgress("' + id + '")', INTERVAL);
```

par celui-ci :

```
eval(requester.responseText);
```

C'est tout ! La délégation, ça a du bon, vous ne trouvez pas ?

Passons maintenant côté serveur, où le code est forcément un peu plus lourd. Au lieu de simplement renvoyer la valeur de progression, nous allons composer le JavaScript à exécuter sur le client. Voici l'intégralité du fichier.

Listing 5-26 Le serveur compose le fragment de JavaScript à exécuter

```
#! /usr/bin/env ruby

require 'webrick'
include WEBrick

progress = 0
colorChanged = false

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/whatsup') do |request, response|
  response['Content-Type'] = 'text/plain'
  progress += rand(5) + 1
  progress = 100 if progress > 100
  script = %{
```

```
    filler.style.width = '#{progress}%';
    percent.firstChild.nodeValue = '#{progress}%';
  }
  if progress > 50 and !colorChanged
    colorChanged = true
    script += %{
      Element.addClassName(percent, 'over50');
    }
  end
  # Arrivé au bout ? Remise à zéro pour la prochaine fois ;-)
  if 100 == progress
    script += %{
      Element.removeClassName(status, 'working');
      Element.addClassName(status, 'done');
    }
    progress = 0
    colorChanged = false
  else
    script += %{
      gProgressTimer = window.setTimeout(
                    ➥ 'checkProgress("' + id + '")', INTERVAL);
    }
  end
  # Nettoyer le script (pensons aux yeux des débogueurs !)
  # --> on vire l'indentation inutile et les lignes vides
  script.gsub!(/^\s+/, '')
  response.body = script
end

trap('INT') { server.shutdown }

srand
server.start
```

La syntaxe `%{…}` encadre des littéraux textuels où la syntaxe `#{…}` permet d'incorporer des portions variables. Cela est fort utile pour manipuler des portions de code, ici JavaScript.

On remarque que notre script repose sur la disponibilité de variables `filler`, `percent` et `status`, qui sont effectivement définies par le script côté client avant d'appeler `eval` sur notre fragment.

Et voilà ! On peut lancer le serveur, afficher http://localhost:8042/, et voir la barre fonctionner comme avant ! Sauf que le JavaScript d'évolution pour chaque étape est composé sur le serveur.

Voici trois exemples de JavaScript retourné : avant 50 %...

```
filler.style.width = '16%';
percent.firstChild.nodeValue = '16%';
gProgressTimer = window.setTimeout('checkProgress("' + id + '")',
INTERVAL);
```

Au premier passage après 50 % :

```
filler.style.width = '51%';
percent.firstChild.nodeValue = '51%';
Element.addClassName(percent, 'over50');
gProgressTimer = window.setTimeout('checkProgress("' + id + '")',
INTERVAL);
```

En arrivant à 100 % :

```
filler.style.width = '100%';
percent.firstChild.nodeValue = '100%';
Element.removeClassName(status, 'working');
Element.addClassName(status, 'done');
```

Vous voyez : rien d'autre que le strict nécessaire.

JSON : l'idéal pour des données structurées spécifiques

JSON signifie *JavaScript Object Notation*. Il s'agit d'une représentation formelle d'objets de complexité quelconque, en utilisant uniquement des syntaxes disponibles en JavaScript, principalement pour décrire des tableaux et des objets. Le format est documenté sur http://www.json.org, qui fournit également des liens vers les principales bibliothèques JSON pour de nombreux langages.

Les bases de JSON sont très simples :

- Un objet est décrit entre accolades (`{}`), sous forme d'une série de paires nom + valeur, séparées par des virgules. Le nom est fourni en tant que chaîne de caractères (donc encadré par `'` ou `"`), et la valeur peut être n'importe quel littéral JavaScript (ou presque) : chaîne de caractères, nombre, booléen, tableau, `null`, etc.
- Un tableau est décrit entre crochets (`[]`), ses éléments sont séparés par des virgules.

Par ailleurs, pour faire dans l'impeccable, nous allons également renvoyer un en-tête de réponse spécialisé, nommé `X-JSON`, qui indique que notre retour est une donnée JSON (il n'existe pas de type de contenu dédié pour l'instant, et cet en-tête est reconnu par un nombre grandissant de bibliothèques, dont Prototype).

Nous allons utiliser comme exemple un affichage de statistiques. Histoire de mélanger un peu les genres, nous allons afficher un pourcentage sans signification particulière pour trois marchés internationaux : le CAC40, le NYSE et le NASDAQ, accompagné d'un commentaire tiré au hasard parmi certaines possibilités.

Nous avons ici plus qu'une simple donnée textuelle ; nous avons plusieurs occurrences d'une même information structurée, laquelle comprend trois parties :

1. le symbole du marché (`cac40`, `nyse` ou `nasdaq`) ;
2. le pourcentage pour ce marché ;
3. le texte de commentaire.

Nous allons donc devoir générer une représentation JSON ressemblant à ceci (l'indentation est juste là pour la lisibilité, elle n'a aucun impact en JavaScript) :

```
[
  { 'symbol': 'cac40', 'percent': 78, 'comment': 'nerveux' },
  { 'symbol': 'nyse', 'percent': 16, 'comment': 'calme' },
  { 'symbol': 'nasdaq', 'percent': 43, 'comment': 'actif' }
]
```

On peut repartir d'un exemple précédent avec barres de progression, car nous allons légèrement ajuster le code HTML et CSS pour correspondre à nos besoins.

Voici d'abord notre serveur.

Listing 5-27 Le serveur envoyant des données JSON

```
#! /usr/bin/env ruby

require 'webrick'
include WEBrick

STOCKS = [ 'cac40', 'nyse', 'nasdaq' ]
COMMENTS = [ 'actif', 'nerveux', 'calme', 'en folie !' ]

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/stats') do |request, response|
  response['Content-Type'] = 'text/plain'
  response['X-JSON'] = true
  data = '['
  STOCKS.each { |symbol|
    comment = COMMENTS[rand(COMMENTS.size)]
```

```
    data += "\n\t{ 'symbol': '#{symbol}', 'percent': #{rand(100)},
            ➥ 'comment': '#{comment}' },"
  }
  data.chop!
  data += "\n]"
  response.body = data
end

trap('INT') { server.shutdown }

srand
server.start
```

Côté client, les barres n'ont plus d'icône ou d'animation à leur droite. Chaque barre a une taille fixe (le remplissage varie évidemment en fonction du pourcentage), et le texte de commentaire prend le reste de la place. On a donc le code HTML suivant.

Listing 5-28 La page HTML client

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Statistiques avec JSON</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>
<h1>Statistiques avec JSON</h1>
<p>Statistiques mises à jour toutes les 2 secondes.</p>
<div class="progressBar" id="progress-cac40">
    <span class="pbFrame">
        <span class="pbColorFill"></span>
        <span class="pbPercentage">0%</span>
    </span>
    <span class="pbComment">n/a</span>
</div>
<div class="progressBar" id="progress-nyse">
    <span class="pbFrame">
        <span class="pbColorFill"></span>
        <span class="pbPercentage">0%</span>
    </span>
    <span class="pbComment">n/a</span>
</div>
```

```
<div class="progressBar" id="progress-nasdaq">
    <span class="pbFrame">
        <span class="pbColorFill"></span>
        <span class="pbPercentage">0%</span>
    </span>
    <span class="pbComment">n/a</span>
</div>
</body>
</html>
```

Et la CSS ajustée que voici.

Listing 5-29 La nouvelle CSS

```
div.progressBar {
    font-family: sans-serif;
    position: relative;
    margin: 1em 0; height: 20px;
}

span.pbFrame {
    position: absolute;
    left: 0; top: 0; width: 20ex; height: 16px;
    border: 2px solid #444;
    background-color: silver;
}

span.pbColorFill {
    position: absolute;
    left: 0; top: 0; height: 100%; width: 0%;
    z-index: 1;
    background-color: green;
}

span.pbPercentage {
    position: absolute;
    left: 0; top: 0; height: 100%; width: 100%;
    z-index: 2;
    font-size: 1em;
    line-height: 16px;
    font-weight: bold;
    text-align: center;
}

span.pbPercentage.over50 {
    color: white;
}
```

```
span.pbComment {
    position: absolute;
    left: 21ex; right: 0; top: 0; line-height: 1.5em;
    color: #444;
    z-index: 2;
}
```

Côté code client, il nous reste à demander toutes les 2 secondes, en mode asynchrone, des nouvelles de nos marchés au serveur. Et bien sûr, à traiter le résultat. On va voir que, JavaScript interprétant le code JSON comme des objets, cela donne un traitement plutôt pratique à écrire.

Listing 5-30 Le script d'invocation et de traitement des statistiques

```
INTERVAL = 2000;
var gLookupTimer = 0;

function getRequester() {
...
} // getRequester

function loadStats() {
    window.clearTimeout(gLookupTimer);
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            var data = $A(eval(requester.responseText)); ❶
            updateStats(data);
            gLookupTimer = window.setTimeout('loadStats()', INTERVAL);
        }
    };
    requester.open('GET', '/stats', true);
    requester.send(null);
} // loadStats

function updateStats(data) {
    data.each(function(stock) { ❷
        var bar = $('progress-' + stock.symbol);
        var filler = document.getElementsByClassName(
                       ➥ 'pbColorFill', bar).first();
        var percent = document.getElementsByClassName(
                       ➥ 'pbPercentage', bar).first();
        var comment = document.getElementsByClassName(
                       ➥ 'pbComment', bar).first();
        filler.style.width = stock.percent + '%';
        percent.firstChild.nodeValue = stock.percent + '%';
        comment.firstChild.nodeValue = stock.symbol.toUpperCase()
                                       ➥ + '' + stock.comment;
```

```
        if (stock.percent > 50)
            Element.addClassName(percent, 'over50');
        else
            Element.removeClassName(percent, 'over50');
    });
} // updateStats

Event.observe(window, 'load', loadStats, false);
```

❶ On convertit le résultat en tableau « boosté » par Prototype et on passe la main à une fonction dédiée de traitement.

❷ `stock` est tour à tour chaque objet statistique. Remarquez comme on a accès directement à ses propriétés `symbol`, `percent` et `comment` !

Au final, on obtient par exemple ceci.

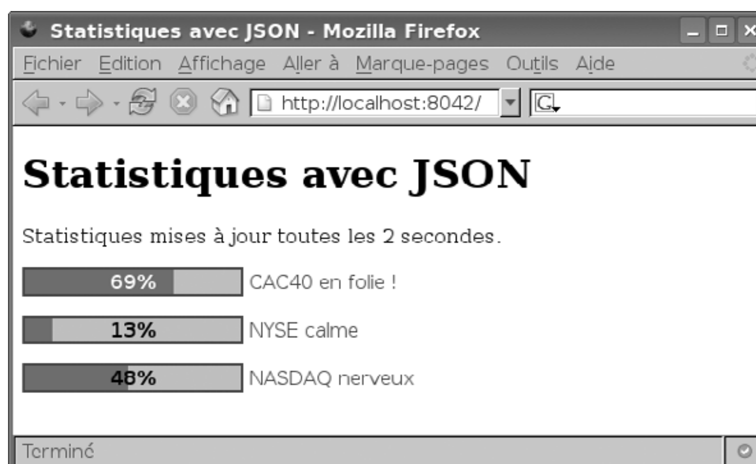


Figure 5–11
L'écran de statistiques

Nous en avons fini avec l'exploration des principaux types de réponse. J'aime à croire que ces exemples auront excité votre imagination et peut-être suggéré des débuts de réponse pour vos besoins réels.

Mais avant de vous jeter dans le code, souvenez-vous que tout ce que nous avons accompli jusqu'ici, nous l'avons réalisé sans tirer pleinement parti de Prototype et sans aucun framework supplémentaire. Attendez donc d'avoir lu les chapitres suivants, qui vont simplifier tout cela et ouvrir encore davantage de perspectives, avant de partir appliquer ces connaissances à vos développements.

6

Ajax tout en souplesse avec Prototype

Nous avons jusqu'ici utilisé Ajax « à la main » : nous avons manipulé `XMLHttpRequest` directement, ce qui engendre une certaine complexité, ou tout au moins un volume de code important, en particulier si on considère la simplicité du service qu'il rend. Dans ce chapitre, nous allons sauter le pas et tirer parti des utilisations prédéfinies d'Ajax proposées par Prototype.

Avant de commencer, je précise aux lecteurs qui auraient la bonne idée de développer avec Ruby on Rails que l'immense majorité des fonctions de Prototype dispose de *helpers* très pratiques à utiliser dans vos vues et *templates* RJS, qui vous évitent de devoir gérer les détails. Reportez-vous à la documentation de la classe `PrototypeHelper`pour plus d'informations (http://api.rubyonrails.org).

Prototype encore à la rescousse

En réalisant les exemples précédents, on s'aperçoit qu'on écrit pas mal de code de « bas niveau » : l'algorithme sinueux d'obtention d'un requêteur, les multiples appels pour paramétrer et envoyer la requête, et ceci de différentes façons suivant qu'on est en `POST` ou en `GET`...

Comme au chapitre 4, Prototype vient à notre rescousse, cette fois-ci pour nous permettre d'écrire facilement du code reposant sur Ajax, sans nous soucier des incompatibilités éventuelles entre navigateurs, et d'une façon plus orientée objet, plus élégante.

Ajax.Request, c'est tellement plus simple !

Prototype fournit une aide considérable à l'utilisation d'Ajax au travers de trois classes principales et de quelques possibilités annexes, un peu plus ésotériques. Nous allons commencer par le comportement de base : l'exécution d'une unique requête et le traitement de sa réponse.

Commençons par une utilisation appliquée, en ajustant notre exemple de sauvegarde automatique, issu du chapitre précédent. Nous attendrons la prochaine section pour dresser un inventaire détaillé des possibilités.

Recopiez votre arborescence de travail pour l'exemple de sauvegarde automatique que nous avions réalisé comme première application d'un type de réponse « texte simple ». Nous allons simplement modifier le script dédié, dans le fichier `client.js`. Ces modifications consistent à :

1. Retirer notre fonction `getRequester` (enfin !).
2. Modifier le code de `syncName` pour tirer parti de Prototype au lieu de récupérer, configurer, exécuter et traiter manuellement notre requête et sa réponse.

Voici le texte intégral de notre nouveau `client.js`.

Listing 6-1 Le nouveau script client pour notre sauvegarde automatique

```
function bindTextField(e) {
    Event.observe($('edtName'), "blur", syncName, false);
} // bindTextField
```

```
function syncName(e) {
    new Ajax.Request('/save_name',
    {
        method:'get',
        parameters: $H{ 'name': $F('edtName') }).toQueryString(),
        onComplete: function(requester) {
            if (this.responseIsFailure() || null ==
                              ➥ requester.responseText.match(/^200/))
                alert('Le nouveau nom n\'a pas pu être sauvegardé !');
        }
    });
} // syncName

Event.observe(window, "load", bindTextField, false);
```

Comme vous le voyez, c'est assez court. Les lecteurs attentifs (donc vous) auront remarqué l'absence d'un appel de type `send()`. La raison est simple : un objet `Ajax.Request` déclenche sa requête immédiatement après initialisation.

Celle-ci comprend deux arguments : l'URL de base à invoquer (sans les paramètres, contrairement à notre code manuel précédent) et un tableau associatif d'options. Celles-ci sont nombreuses, mais contentons-nous pour l'instant d'expliciter les trois que notre exemple utilise :

- `method` permet d'indiquer la méthode HTTP à utiliser pour la requête. Par défaut à `post`, elle est ici modifiée en `get`, conformément aux exigences de notre couche serveur, qui n'a pas changé.
- `parameters` permet de préciser des paramètres pour une requête de type `get`, qui seront placés dans la *query string* de l'URL (le point d'interrogation est ajouté automatiquement). Cette valeur doit être encodée.
- `onComplete` fournit une fonction de rappel qui sera invoquée une fois la réponse terminée (quel que soit l'état du requêteur).

Dans ces fonctions de rappel, `this` est associé à l'objet `Ajax.Request` qui les invoque. C'est pourquoi on dispose, entre autres, d'une méthode `responseIsFailure()`, qui permet de déterminer qu'un problème a été rencontré par le requêteur.

À part ça, rien ne change ! Sur un exemple aussi simple, on ne se rend pas bien compte de la puissance offerte par `Ajax.Request`. Aussi, la section suivante explore cette classe plus en détail, afin de vous mettre l'eau à la bouche...

Plus de détails sur Ajax.Request

Techniquement, cette classe hérite de `Ajax.Base`, ce qui permet à ses objets de récupérer trois méthodes.

Méthode	Description
`responseIsFailure()`	Méthode à n'appeler qu'après la fin de la requête, qui détermine si un problème a été rencontré. L'exact inverse de `responseIsSuccess()`.
`responseIsSuccess()`	Méthode à n'appeler qu'après la fin de la requête, qui vérifie que la requête s'est bien passée, c'est-à-dire que la propriété `status` du requêteur est soit indéfinie, soit nulle (zéro), soit dans la tranche 200-299.
`setOptions(options)`	Permet de définir les options de requête, en définissant d'abord les valeurs par défaut décrites plus bas, puis en appliquant celles fournies sous forme d'un tableau associatif ou d'un objet.

Tous les requêteurs Ajax de Prototype partagent un certain nombre d'options. En voici la liste.

Tableau 6–1 Les options disponibles pour Ajax.Request et ses versions spécialisées

Option	Défaut	Description
`asynchronous`	`true`	Mode de synchronisation de la requête. On n'y touche généralement pas.
`method`	`'post'`	Méthode HTTP à utiliser. Valeurs possibles garanties : `'get'` et `'post'`. Les standards en cours d'élaboration exigent néanmoins aussi le support de `'put'`, `'delete'` et `'head'`, notamment pour faciliter l'accès Ajax à des API de type REST. La casse est sans importance.
`onLoaded`, `onInteractive`, `onComplete`	`undefined`	Fonctions de rappel pour les états « envoyé », « en réception » et « réception terminée », respectivement. Cela permet de séparer le code affecté à chaque étape du traitement, plutôt que d'avoir à tester la propriété `readyState` du requêteur. On n'utilise toutefois que rarement les deux premières.
`onSuccess`, `onFailure`, `on`*XYZ*	`undefined`	Il est possible de distinguer encore plus avant en séparant le code de traitement après requête couronnée de succès, de celui de réaction en cas de problème. On peut même associer un rappel à un code de retour précis (par exemple, 302, 201, etc. Il suffit de définir une association nommée d'après le code, comme `on302`). Ces fonctions sont appelées avant (et non à la place de) une éventuelle fonction associée à `onComplete`.
`onException`	`undefined`	Gestionnaire d'exceptions dans la couche client de requêtage. Prend deux arguments : l'objet `Ajax.Requester` qui a rencontré un problème, et l'exception JavaScript qui a été levée.

Tableau 6–1 Les options disponibles pour Ajax.Request et ses versions spécialisées (suite)

Option	Défaut	Description
`parameters`	`''`	Paramètres URL encodés pour une requête GET (sans le ? initial).
`postBody`	`undefined`	Corps de requête pour une requête `POST`. Ce sera généralement de l'URL encodé aussi.

`requestHeaders`	`undefined`	Permet de définir des en-têtes de requête, par exemple un en-tête `Accept` pour demander un type de contenu réponse particulier. Il doit s'agir d'un tableau au nombre d'éléments pair, constitué de couples nom/valeur. Par exemple, pour demander un résultat XML, on pourrait faire :
		`new Ajax.Request('/mon_url_cible', {`
		`    ...`
		`    **requestHeaders: [ 'Accept',**`
		`    ➥ **'application/xml' ],**`
		`    onSuccess: function(req) {`
		`        // traitement de req.responseXML`
		`    }`
		`});`
		Ne pas confondre avec la méthode `header`, décrite plus bas, qui permet de consulter les en-têtes émis dans la réponse.

Information parfois utile : tout objet `Ajax.Request` dispose, une fois le requêteur interne créé, d'une propriété `transport` qui référence ce requêteur.

Autre point notable : si la réponse est renvoyée avec un type MIME `text/javascript`, son contenu texte sera considéré comme un fragment de JavaScript à évaluer, et sera donc évalué automatiquement. Pour les curieux, sachez que cette évaluation a lieu après les éventuels `onSuccess`, `onFailure` ou `on`*XYZ*, mais avant l'éventuel `onComplete`.

En parlant de contenus particuliers, sachez que chaque fonction de rappel (`onException` n'est pas concernée) reçoit deux arguments : d'abord l'objet `XMLHttpRequest` en cours d'utilisation, ensuite un booléen indiquant si le contenu renvoyé est signalé comme contenu JSON (au moyen de l'en-tête de réponse `X-JSON`). Ce deuxième argument n'a toutefois de sens qu'une fois la requête terminée avec succès, donc dans `onSuccess` et éventuellement `onComplete`. S'il est vrai, vous pouvez donc évaluer le texte de réponse. Petite astuce au passage : vous pouvez vous simplifier légèrement la tâche en écrivant :

```
this.evalResponse()
```

au lieu de :

```
eval(req.responseText)
```

Les deux renvoient le résultat de l'évaluation, ce qui est bien pratique pour récupérer l'objet dont la représentation JSON a été fournie en réponse.

Enfin, `Ajax.Request` fournit une méthode supplémentaire, `header(`*`name`*`)`, qui renvoie l'en-tête de réponse demandé. Cette méthode n'a donc d'utilité qu'une fois la requête terminée. Elle correspond à la méthode `getResponseHeader` de `XMLHttpRequest`.

Voilà, à présent vous savez tout sur `Ajax.Request` !

Ajax.Updater : mettre à jour un fragment XHTML, exécuter un script

Évidemment, Prototype ne s'arrête pas là (ce n'est pourtant déjà pas mal !). Il fournit d'autres mécanismes, dédiés à des utilisations récurrentes d'Ajax. L'un d'eux est fourni par la classe `Ajax.Updater`, qui est spécialisée dans l'envoi de requêtes dont la réponse est censée mettre la page à jour. Il s'agit naturellement d'une classe fille de `Ajax.Request`.

Néanmoins, le constructeur n'est plus tout à fait le même : avant les arguments d'URL et d'options, `Ajax.Updater` prend un premier argument désignant l'élément qui fera l'objet d'une mise à jour. Comme la plupart du temps avec Prototype, cet argument peut être une `String` contenant l'ID de l'élément, ou l'élément lui-même. On verra un peu plus tard qu'il existe aussi une autre forme, plus avancée, pour ce premier argument.

Remarquez également que dans la mesure où le traitement de la réponse est entièrement automatique, définir des fonctions de rappel telles que `onComplete` ou `onSuccess` n'a pas beaucoup d'intérêt… S'il ne s'agit alors que de traitements génériques à toutes vos requêtes Ajax (journalisation, etc.), il existe de meilleurs moyens, que nous verrons un peu plus loin.

Armés des connaissances acquises dans le précédent chapitre, nous savons que les deux grands cas de figure pour une réponse modifiant la page sont : XHTML et JavaScript. Il y a donc déjà une distinction potentielle sur le type du contenu renvoyé.

Il y a aussi diverses possibilités sur le type de la mise à jour : s'agit-il de remplacer un fragment du document (une valeur de pourcentage par exemple, qui aura peut-être évolué), ou d'ajouter au document ? Et dans ce deuxième cas, où l'insertion doit-elle avoir lieu ?

Malgré l'apparente complexité engendrée par la combinaison de ces possibilités, Prototype nous permet de préciser très simplement nos besoins à l'aide de deux options supplémentaires.

Tableau 6–2 Options supplémentaires pour Ajax.Updater

Option	Défaut	Description
`evalScripts`	`false`[a]	Active l'évaluation automatique de scripts (c'est-à-dire de code JavaScript). Là, vous êtes tout surpris, car je vous ai expliqué tout à l'heure que `Ajax.Request` (et donc `Ajax.Updater`) évaluent automatiquement les scripts JavaScript renvoyés. C'est juste, mais uniquement si la réponse affiche un type MIME `text/javascript`. Afin de permettre à ceux qui ne pourraient pas manipuler les en-têtes dans la réponse HTTP de bénéficier de cette possibilité, Prototype offre cette option, dont l'activation préservera les portions `<script…>…</script>` du corps de la réponse, au lieu de les supprimer. Le corps final étant inséré dans le document, le script ainsi fourni sera effectivement évalué ! Notez au passage qu'en laissant cette option désactivée, Prototype vous protège contre des attaques malicieuses à base de script dans le contenu renvoyé (par exemple dans le cadre d'une syndication de contenu).
`insertion`	`undefined`	Toute la flexibilité de `Ajax.Updater` vient de là ! En laissant cette option indéfinie, vous causerez le remplacement du contenu de l'élément désigné via le constructeur. C'est idéal pour mettre à jour un texte ou même un fragment DOM à remplacer complètement, par exemple tout ou partie d'un graphique SVG (ce qui est assez glamour, je trouve). Mais que faire lorsqu'il s'agit de compléter le document, par exemple pour ajouter un commentaire ou un envoi sur un forum en ligne « live » ? Il suffit de préciser ici l'une des classes d'insertion fournies par Prototype, c'est-à-dire l'une des classes définies à l'intérieur de l'espace de noms `Insertion`. Vous en retrouverez la liste dans le tableau suivant.

a. En réalité, undefined, ce qui est plus cohérent pour marquer l'absence de définition. Mais comme vous le savez, évalué comme un booléen, cela équivaut à false.

Nous évoquons dans ce tableau le mécanisme des insertions, qui permet de ne pas remplacer le contenu de l'élément, mais plutôt d'ajouter du contenu. Ces ajouts dépendent de la classe utilisée pour l'insertion et Prototype fournit des classes adaptées à chaque besoin. Nous les avons déjà étudiées au chapitre 4, mais je vous offre un petit rappel. Les résultats présentés dans le tableau ci-dessous supposent le fragment XHTML initial suivant :

```
<p>
    <span id="colmin">Graff</span>
</p>
```

Et l'exécution ultérieure d'un code de type :

```
new Insertion.LaClasseDInsertion('colmin', ' <em>Hyrum</em> ')
```

Voici les classes d'insertion disponibles.

Tableau 6–3 Classes d'insertion pour l'ajout de contenu au DOM

Classe	Emplacement de l'insertion	Résultat
Insertion.After	Immédiatement après l'élément.	 Graff **Hyrum**
Insertion.Before	Immédiatement avant l'élément.	 **Hyrum** Graff
Insertion.Bottom	Après le contenu de l'élément.	 Graff **Hyrum**
Insertion.Top	Avant le contenu de l'élément.	 **Hyrum** Graff

À présent que nous avons couvert à nouveau la théorie, voyons comment adapter notre exemple de réponse XHTML. Copiez votre répertoire de travail pour l'exemple de réponse XHTML et modifiez le fichier `client.js` : retirez notre fonction `getRequester` et modifiez `switchToAJAX` pour obtenir le code suivant.

Listing 6-2 Le nouveau script client pour notre mise à jour XHTML

```
function bindForm() {
    Event.observe($('commentForm'), 'submit', switchToAJAX, false);
} // bindForm

function switchToAJAX(e) {
    Event.stop(e);
    var data = $H({
        'name': $F('edtName'),
        vemail': $F('edtEmailv),
        'comment': $F('memComment')
    }).toQueryString();
    new Ajax.Updater('comments', '/add_comment',
    {
        postBody: data,
        insertion: Insertion.Bottom
    });
```

```
} // switchToAJAX

Event.observe(window, 'load', bindForm, false);
```

Avouez que c'est impressionnant ! Vous pouvez lancer votre script `serveur.rb`, afficher http://localhost:8042 et rafraîchir en forçant le contournement du cache, vous verrez : ça marche toujours (heureusement, ceci dit !). C'est « juste » plus court et plus élégant.

Inutile de présenter un exemple d'utilisation de l'option `evalScripts`. D'une part, la plupart des technologies côté serveur que vous pourriez utiliser vous permettent de modifier les en-têtes de réponse : un simple `Ajax.Request` sur une réponse de type `text/javascript` suffira donc amplement. D'autre part, la mise en œuvre de `evalScripts` est, vraiment, tellement simple qu'elle se passe d'exemple ! Il suffit d'encoder son JavaScript pour éviter les chevrons (< et >) littéraux et d'encadrer le tout par `<script type="text/javascript">` et `</script>`.

Différencier la mise à jour entre succès et échec

Lorsque j'ai présenté `Ajax.Updater`, j'ai précisé qu'il existait une forme plus avancée pour le premier argument du constructeur. En effet, tel que nous l'avons utilisé jusqu'à présent, il effectue la mise à jour même si la requête a échoué (c'est-à-dire si, dans la fonction de rappel `onComplete`, la méthode `responseIsFailure()` renvoie `true`) ! Ce n'est pas forcément ce qu'on veut.

On a deux comportements alternatifs possibles :

- On souhaite une mise à jour en cas de réussite uniquement.
- On souhaite des mises à jour différentes suivant que la requête a réussi ou échoué (par exemple, on a un fragment de page dédié aux messages d'erreur, message que nous renverrait la couche serveur en cas de problème).

Ces deux cas de figure ont recours au même mécanisme : il s'agit de ne pas passer, comme premier argument au constructeur, la désignation d'un seul élément, mais plutôt de lui passer un objet désignant les éléments en cas de succès et d'échec.

Cet objet, généralement anonyme, a simplement besoin de fournir une désignation (l'élément lui-même ou une `String` avec son ID) dans sa propriété `success`, et éventuellement une seconde dans sa propriété `failure`.

- Si vous ne fournissez qu'une propriété `success`, l'échec ne donnera lieu à aucune mise à jour.
- Si vous fournissez les deux, l'échec mettra à jour l'élément désigné par la propriété `failure`.
- Il ne sert à rien (et il serait par ailleurs illogique) de ne préciser qu'une propriété `failure`…

Voici un exemple classique :

```
new Ajax.Updater({ success: 'cartItems', failure: 'errorMsg' },
  '/purchase',
  {
      postBody: lineItemData,
      insertion: Insertion.Bottom
  });
```

Presque magique : Ajax.PeriodicalUpdater

Si `Ajax.Updater` vous a plu, vous allez adorer `Ajax.PeriodicalUpdater` ! Attention, la seconde n'est pas une classe fille de la première. En effet, il ne s'agit pas d'un type spécialisé de requêteur, mais d'un mécanisme de répétition intelligent autour d'une utilisation normale de `Ajax.Updater`.

Un objet `Ajax.PeriodicalUpdater` se construit strictement comme un objet `Ajax.Updater`, mais dispose de deux options supplémentaires.

Tableau 6–4 Options supplémentaires pour Ajax.PeriodicalUpdater

Option	Défaut	Description
`frequency`	2	Période (oui, Sam Stephenson n'est pas très au point sur le couple fréquence/période...), en secondes, entre deux invocations de `Ajax.Updater`. Évitez de préciser de trop petites valeurs (par exemple `0.1` alors que vous n'êtes pas en intranet et sur un serveur rapide), sans quoi vous risqueriez, au mieux, d'obtenir un comportement incohérent, au pire, de geler le navigateur, voire la machine.
`decay`	1	(Prononcez « di-kay ») Signifie littéralement *décomposition*, *décrépitude*, *déliquescence*... Et pourtant, c'est une fonctionnalité qui, pour être avancée, n'en est pas moins précieuse. Elle permet de demander un allongement progressif de la période (donc, pour les pointilleux, une baisse de la fréquence) lorsque le résultat ne varie pas d'une requête à l'autre. Explications plus en détail à la prochaine section.

Autre précision de taille : une méthode `stop()` permet d'arrêter le cycle d'invocations. S'il s'agit d'un arrêt temporaire, vous pouvez relancer le traitement ultérieurement avec la méthode `start()`, appelée automatiquement après l'initialisation de l'objet.

Par ailleurs, la fonction de rappel `onComplete` a ici une utilité, contrairement au cas de `Ajax.Updater`. En effet, la fonction de rappel serait invoquée à la fin de la méthode `stop()` (et non en fin de réception de réponse Ajax).

Comprendre l'option decay

Voici comment l'option `decay` fonctionne : à chaque requête dont le résultat est identique à celui de la précédente, la valeur active de `decay` (qui démarre à celle de l'option) est multipliée par la valeur de l'option. La période avant la prochaine requête est alors définie en multipliant la période (option `frequency`) par le `decay` actif. En revanche, dès qu'une requête ramène un résultat différent, le `decay` actif repasse à 1 (un).

Donc, une option `decay` de 1 (un) revient à désactiver cette fonctionnalité. C'est le cas par défaut. Et une option `decay` inférieure à un serait contre-productive : alors que rien n'a l'air de se passer côté serveur, on demanderait de plus en plus souvent ce qui se passe !

Imaginons en revanche une option `decay` à 2 (histoire de simplifier les valeurs, parce qu'en réalité, on opte plutôt pour une valeur entre 1,1 et 1,5 dans la plupart des cas). Voici un petit tableau résumant une séquence d'appel.

Tableau 6–5 Exécution d'un Ajax.PeriodicalUpdater, option decay à 2

Moment	Decay actif	Période	transport. responseText	Réaction
00:00	2	2	42	Réponse différente (pas d'antérieur) : *decay*1.
00:02	1	2	54	Idem.
00:04	1	2	54	Réponse identique : *decay* × option `decay`, et comme période = option `frequency` × *decay*…
00:08	2	4	57	Réponse différente : *decay* 1.
00:10	1	2	57	Même comportement qu'à 00:04.
00:14	2	4	57	C'est le début du grand ralentissement…
00:22	4	8	57	…
00:38	8	16	57	32 secondes avant la prochaine requête !
01:10	16	32	63	Ah, enfin ! On retombe à *decay* 1.
01:12	1	2	64	…
01:14	1	2	66	Et visiblement, le serveur s'est « décoincé » !

Petits secrets supplémentaires

Tant qu'à explorer les possibilités Ajax de Prototype, autant ne pas se limiter aux utilisations courantes (je parie que vous aussi, vous aimez en savoir plus que les autres), non ? Jetons donc un coup d'œil sur quelques points moins fréquemment mis en lumière.

Commençons avec deux petits détails : l'espace de noms `Ajax` fournit, en plus des trois classes que nous venons d'explorer, une méthode `getTransport()` et une propriété `activeRequestCount`.

La première est sans mystère : `Ajax.getTransport()` est la méthode appelée, en interne, par les classes de requête pour obtenir le requêteur, c'est-à-dire l'objet `XMLHttpRequest` à proprement parler. C'est l'équivalent de notre bonne vieille fonction `getRequester`, mais en tellement plus joli :

Listing 6-3 Ne vous avais-je pas dit que getTransport() était esthétique ?

```
return Try.these(
      function() {return new ActiveXObject('Msxml2.XMLHTTP')},
      function() {return new ActiveXObject('Microsoft.XMLHTTP')},
      function() {return new XMLHttpRequest()}
    ) || false;
```

Décidément, `Try.these`, c'est bien pratique... Et ce code n'est que marginalement plus lent que notre `getRequester`.

La propriété `Ajax.activeRequestCount` est tenue à jour automatiquement par Prototype et fournit le nombre de requêteurs créés mais dont le traitement n'est pas encore terminé. On imagine l'utilité éventuelle de cette propriété dans un contexte de journalisation ou de surveillance de l'activité Ajax de la page.

Et cependant...

Puisque `getTransport()` se contente de renvoyer un objet `XMLHttpRequest`, objet sur lequel il n'a pas de contrôle particulier, comment diable Prototype fait-il pour être tenu au courant de la fin de traitement des requêteurs ? (Vous vous êtes bien posé la question, n'est-ce pas ?)

Si vous êtes comme moi, vous n'aimez pas l'impression de boîte noire qu'on a en découvrant un nouveau framework. Dissipons donc rapidement le voile de mystère qui plane sur cet apparent tour de force : Prototype utilise tout simplement le dernier secret dont je voulais vous parler : `Ajax.Responders`.

`Ajax.Responders` est un objet global qui fournit un point centralisé d'inscription pour des fonctions de rappel globales à tous les requêteurs. En d'autres termes, une fonction de rappel enregistrée auprès de `Ajax.Responders` s'ajoute à celles définies pour tout `Ajax.Request` ou `Ajax.Updater` (et par conséquent, tout `Ajax.PeriodicalUpdater` également).

Ce qui signifie, incidemment, que si vous obtenez un requêteur manuellement en appelant `getTransport`, il n'est pas concerné (remarquez bien que vous n'avez pas de raison particulière de bouder ainsi les mécanismes standards).

Prototype se contente donc d'enregistrer d'office auprès de `Ajax.Responders` deux fonctions de rappel, pour `onCreate` et `onComplete`, qui maintiennent notre fameux `activeRequestCount` à jour. C'est tout simple !

En ce qui nous concerne, `Ajax.Responders` a deux méthodes intéressantes : `register` et `unregister`, dont les noms sont plutôt explicites. Chacune prend en paramètre un objet dont les méthodes ont les noms des rappels souhaités. Il peut s'agir d'objets anonymes, comme justement dans Prototype :

```
Ajax.Responders.register({
  onCreate: function() {
    Ajax.activeRequestCount++;
  },

  onComplete: function() {
    Ajax.activeRequestCount--;
  }
});
```

Remarquez qu'ici, Prototype n'ayant que faire des détails des requêteurs, il ne nomme pas les paramètres pourtant passés aux fonctions de rappel : le requêteur (`XMLHttpRequest`) et le statut JSON de la réponse (booléen, déjà évoqué plus haut).

Les fonctions de rappel globales ainsi définies sont appelées immédiatement après les fonctions de rappel spécifiques à chaque requêteur (telles que celles que nous avions définies jusqu'à présent).

Notez toutefois que les rappels spéciaux `onSuccess`, `onFailure` et `on`*`XYZ`* (rappels numériques) ne participent pas à ce mécanisme : il ne peut pas y avoir de tels rappels globaux. Seuls les rappels calés sur le cycle de vie d'un requêteur, ainsi que les gestionnaires d'exceptions, sont éligibles à un traitement global.

Enfin, pour la petite histoire, sachez que `Ajax.Responders` est un `Enumerable`. Si cela ne vous dit rien, prêtez donc l'oreille : entendez-vous le champ de sirène du chapitre 4 ? Ne lui résistez pas ! Je vous attendrai sagement ici.

Voilà, croyez-le ou non, entre le chapitre 4 et celui-ci, nous avons fait une exploration raisonnablement complète de Prototype (en somme, nous avons laissé de côté les détails internes et l'objet `Position`, mais pour le reste, nous avons vu tout ce qui pouvait nous être utile).

Voici donc une des promesses de ce livre, et plus particulièrement de ce chapitre, qui est tenue. Mais je ne vous ai pas appâté jusqu'ici sur la base de cette seule promesse, j'ai aussi fait allusion au monde merveilleux des effets visuels riches et d'interfaces utilisateurs très dynamiques, avec du glisser-déplacer, de la complétion automatique de texte, et autres merveilles... Il est temps d'assumer ces allusions. Le chapitre 7 va vous plonger dans script.aculo.us.

Pour aller plus loin...

Livres

Pragmatic Ajax: A Web 2.0 Primer
Justin Gehtland, Ben Galbraith, Dion Almaer
Pragmatic Programmers, mars 2006, 296 pages
ISBN 0-9766940-8-5

Dont la version française est :

Ajax par la pratique
O'Reilly France, mai 2006, 250 pages
ISBN 2-84177-387-6

Sites

On compte aujourd'hui de nombreuses communautés Ajax établies. Pour n'en citer que trois :

- **Ajaxian** : http://ajaxian.com
- **AjaxPatterns** : http://ajaxpatterns.org
- **Ajax Developer's Journal** : http://ajaxdevelopersjournal.com

Baekdal.com, le site de Thomas Baekdal. Vous y trouverez de nombreux articles de bonne qualité (voire excellents) autour d'Ajax et des technologies Web 2.0 : http://baekdal.com.

Groupe de discussion

Le groupe Google RubyOnRails-Spinoffs a été créé en août 2006 pour servir de centre d'aide technique, avec les avantages d'indexation et de recherche de Google. On y trouve de nombreux membres avec un bon niveau technique.

http://groups.google.com/group/rubyonrails-spinoffs

Canal IRC

Enfin, il faut noter qu'on trouve souvent une réponse rapide et fiable sur le canal IRC dédié à Prototype, hébergé sur l'incontournable serveur `irc.freenode.net`. Le canal se nomme tout simplement `#prototype`.

7

Une ergonomie de rêve avec script.aculo.us

Il est temps d'explorer l'univers des effets visuels, des comportements avancés, et de leur impact ergonomique avec script.aculo.us. Nous prendrons aussi le temps de réfléchir à la problématique, parfois subtile, de l'accessibilité des interfaces utilisant Ajax. Nous conclurons avec un tour d'horizon des principaux frameworks que nous n'aurons pas eu la place de traiter.

Avant de commencer, je précise aux lecteurs qui auraient la bonne idée de développer avec Ruby on Rails que l'immense majorité des fonctions de script.aculo.us dispose de *helpers* très pratiques à utiliser dans vos vues et *templates* RJS, qui vous évitent de devoir gérer les détails. Reportez-vous à la documentation de la classe `ScriptaculousHelper` pour plus d'informations (http://api.rubyonrails.org).

Une ergonomie haut de gamme avec script.aculo.us

Rendons visite à Thomas Fuchs, le jeune Autrichien auteur de script.aculo.us (entre autres jolies choses).

Cette bibliothèque JavaScript réussit le véritable tour de force de mettre à disposition de tout un chacun, sous forme de petits objets JavaScript faciles d'emploi, des effets visuels et éléments d'interfaces utilisateur complexes, et cela, sur la plupart des navigateurs (comprendre : sur MSIE aussi !). Chapeau bas, Herr Fuchs !

Mais je sens bien que je ne saurais vous convaincre sur la seule base de mon enthousiasme débordant. Dans ce cas, permettez-moi de vous montrer.

Commencez par récupérer script.aculo.us : vous la trouverez à la racine de l'archive des codes source, ou sur l'excellent site de la bibliothèque, rempli de documentations, exemples, démonstrations, etc. : http://script.aculo.us. Choisissez l'onglet downloads et prenez la version la plus récente (1.6.4 à l'heure où j'écris ces lignes), dans le format d'archive que vous préférez. Vous y piocherez les fichiers JS dont nous aurons besoin (pas tous !) au fil de nos exemples, dans les répertoires `lib` et `src` de l'archive.

> **QUALITÉ Des tests à foison !**
>
> script.aculo.us montre l'exemple en prouvant avec brio qu'il est parfaitement possible d'appliquer une méthodologie de test rigoureuse à du code JavaScript. La bibliothèque est dotée de plus de 80 tests unitaires regroupant environ 1 000 assertions, exécutables et consultables de façon automatisée. Plusieurs dizaines de tests fonctionnels sont également maintenus.
> Ouvrez par exemple la page `test/run_unit_tests.html` dans votre navigateur et exécutez les tests unitaires. Vous allez être impressionnés !

Charger script.aculo.us

On trouve une petite innovation intéressante dans script.aculo.us, rien que dans la façon dont la bibliothèque se charge. Plutôt que de mettre en place un énorme fichier JS monolithique, qui aurait un impact considérable sur le chargement et éventuellement la consommation mémoire, script.aculo.us est découpée en plusieurs fichiers.

Mais ceci, en soi, n'est pas innovant du tout. Ce qui est inédit, c'est le confort qui nous est offert pour exprimer nos souhaits de chargement parmi ces modules. Plutôt que de nous imposer un élément `<script>` par module (en veillant à placer le noyau, `scriptaculous.js`, en premier), une syntaxe spéciale de chargement est prise en charge par la bibliothèque : il suffit de passer un paramètre `load` à notre fichier noyau, qui contient une liste des modules complémentaires qui nous intéressent, séparés par des virgules. Par défaut, tous les modules sont chargés : `builder`, `effects`, `dragdrop`, `controls` et `slider`. Mais si, comme nous dans un instant, vous

n’avez besoin que d’une partie, précisez les modules concernés dans votre balise de chargement. Par exemple :

```
<script type="text/javascript" src="scriptaculous.js?load=effects">
</script>
```

Plus rapide, plus léger... Mieux, quoi.

Les effets visuels

Nous allons d’abord examiner les effets visuels. Ils sont découpés en trois catégories :

- Les effets dits « noyau » (*core effects*), qui sont fondamentaux.
- Les effets dits « combinés », qui associent des effets noyau et ajoutent parfois du comportement supplémentaire pour obtenir des effets avancés.
- Les effets du « coffre au trésor », fournis par des tiers sur le site de script.aculo.us, qui sont une catégorie non officielle d’effets combinés.

Nous n’irons pas jouer avec cette dernière catégorie, et nous ne verrons pas en détail tous les effets officiels, mais nous allons tout de même en explorer plusieurs, et décrire brièvement les autres.

Avant de réaliser quelques exemples, nous allons explorer les effets noyau et les concepts généraux d’utilisation des effets. La prochaine section contient de nombreuses informations de référence.

Les effets noyau

Tous les effets, noyau comme combinés, sont fournis sous forme de classes disponibles dans l’objet global `Effect`. On compte 5 effets noyau dans script.aculo.us, sur lesquels se basent tous les autres.

Tableau 7–1 Effets noyau de script.aculo.us

Classe	Description
`Effect.Opacity`	Modifie l’opacité (ou la transparence, suivant le point de vue) de l’élément. Une opacité de 0 % est une transparence totale (position `0.0` de l’effet). Attention, dans MSIE, l’élément doit avoir un *layout* pour bénéficier de cet effet. Si vous utilisez une vieille version de script.aculo.us, mettez-vous à jour ou consultez la page http://wiki.script.aculo.us/scriptaculous/show/GivingElementsLayout pour voir comment gérer ce détail.
`Effect.Scale`	Ajuste progressivement la taille de l’élément et de son contenu jusqu’à un pourcentage donné. Dispose de nombreuses options spécifiques.

Tableau 7–1 Effets noyau de script.aculo.us (suite)

Classe	Description
`Effect.MoveBy`	Déplace l'élément d'un certain nombre de pixels, verticalement et horizontalement. L'élément doit être positionné (`absolute` ou `relative`) pour que l'effet fonctionne sur MSIE.
`Effect.Highlight`	Classiquement connu sous le nom *Fade To Yellow*, cet effet réalise un fondu enchaîné de couleurs de fond pour l'élément. Idéal pour appeler l'attention de l'utilisateur sur un élément fraîchement ajouté ou modifié par un traitement Ajax. Nous verrons plus bas quelques précautions d'emploi.
`Effect.Parallel`	Constitue la base de réalisation des effets combinés, en permettant de synchroniser plusieurs effets (noyau ou combiné), généralement sur un même élément. Nous verrons dans un exemple que sa syntaxe d'invocation est particulière.

Invocation de l'effet

Lancer un effet utilise toujours le même style de syntaxe (sauf pour un `Parallel`) :

```
new Effect.NomDeLEffet(element[, parametres_requis][, options]);
```

Comme d'habitude, le premier argument peut être un ID ou l'élément lui-même. Tant les paramètres que les options sont conceptuellement des *hashes*, donc des objets anonymes la plupart du temps, avec une propriété par paramètre ou option. Enfin, notez le `new` au début, qui est trop souvent omis, mais devient vite obligatoire lorsqu'on souhaite déclencher plusieurs effets en parallèle.

Les paramètres requis varient d'un effet à l'autre ; dans certains cas, il n'y a aucun paramètre requis : on peut alors s'en passer et ne fournir que d'éventuelles options.

Options communes à tous les effets noyau

Tout effet noyau reconnaît les options suivantes.

Tableau 7–2 Options communes à tous les effets noyau

Option	Description
`duration`	Durée de l'effet, en secondes. Nombre à virgule. `1.0` par défaut.
`fps`	Nombre d'ajustements par seconde (*frames per second*). 25 par défaut, très largement suffisant. Limité à 100, mais je ne vois vraiment pas quel intérêt on aurait à dépasser environ 50, seuil absolu de la vision humaine.
`delay`	Attente, en secondes, avant le démarrage de l'effet. `0.0` par défaut.

Tableau 7–2 Options communes à tous les effets noyau (suite)

Option	Description
`transition`	Fonction assurant la progression de l'effet dans le temps (succession des valeurs entre `from` et `to`). Il existe 8 transitions prédéfinies, toutes dans `Effect.Transition` : – `sinoidal` (par défaut) augmente la vitesse de l'effet au fil de son déroulement ; – `linear` maintient une vitesse fixe, souvent moins élégante ; – `reverse` inverse le déroulement de l'effet : il s'effectue de sa fin à son début. ; – `wobble` amène un comportement « dessert en gelée » à l'évolution de l'effet lui-même, très visible sur, par exemple, `Effect.Scale` ; – `flicker` donne une impression de clignotement basée sur les 25 % finaux de l'effet ; – `pulse` accélère un effet de base pour le répéter 5 fois au court de sa durée prévue (utilisé par exemple pour réaliser l'effet `Pulsate`) ; – `none` laisse l'élément dans son état en début d'effet ; – `full` amène immédiatement l'élément en fin d'effet.
`from`	Position au sein de l'effet en début de traitement. Nombre entre `0.0` (0 %) et `1.0` (100 %). `0.0` par défaut.
`to`	Même chose, mais pour la fin du traitement. `1.0` par défaut.
`sync`	Détermine si la progression est automatique (`true`, par défaut), ou s'il faut manuellement progresser (`false`) en appelant `render()` sur l'effet. Utilisé en particulier dans le cadre d'un `Effect.Parallel` pour gérer plusieurs effets s'exécutant de façon synchronisée.
`queue` (depuis la version 1.5)	Permet de placer l'effet courant dans une file d'effets. Peut être un nom de file (`'parallel'` par défaut) ou un objet spécial. Voir la section dédiée aux files d'effets plus loin dans ce chapitre.

Fonctions de rappel

En plus des options à proprement parler, le troisième argument du constructeur peut également être utilisé pour associer des fonctions de rappel sur l'effet (ce dont nous avons rarement besoin nous-même, mais qui est fort utile pour la création d'effets combinés). Les noms disponibles sont `beforeStart`, `beforeUpdate`, `afterUpdate` et `afterFinish`, et parlent d'eux-mêmes. Les fonctions sont appelées avec l'objet `Effect` en argument.

Qu'y a-t-il dans un objet d'effet ?

Justement, qu'y a-t-il dans un objet `Effect` ? Les propriétés suivantes peuvent s'avérer utiles.

Tableau 7–3 Les propriétés utiles d'un objet Effect

Propriété	Description
`element`	L'élément (nœud DOM) sur lequel s'applique l'effet.
`options`	Les options fournies.

Tableau 7–3 Les propriétés utiles d'un objet Effect

Propriété	Description
`currentFrame`	Le numéro du *frame* courant dans l'évolution de l'effet.
`startOn`	Le moment de départ de l'effet (représentation en millisecondes après l'invocation).
`finishOn`	Le moment de fin de l'effet (idem).
`effects[]`	Pour un `Effect.Parallel`, la liste des effets simultanés appliqués.

Et si on essayait quelques effets ?

Mais bien sûr ! D'ailleurs, nous n'aurons même pas besoin d'une couche serveur pour nos tests, tout peut se faire côté client, en JavaScript, avec un doigt de XHTML quand même...

Créez un répertoire de travail, par exemple `script.aculo.us`, ainsi qu'un premier sous-répertoire `opacity`, et placez à l'intérieur, en plus des fichiers `prototype.js`, `scriptaculous.js` et `effects.js` de script.aculo.us, les fichiers suivants, que nous adapterons pour nos essais ultérieurs.

Listing 7-1 Notre index.html pour tester Effect.Opacity

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Test de Effect.Opacity</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="scriptaculous.js?load=effects">
    </script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test de <code>Effect.Opacity</code></h1>

<p id="byebye">Cette page teste l'effet Effect.Opacity. Cliquez sur ce
paragraphe pour avoir une démonstration.</p>

</body>
</html>
```

Listing 7-2 Notre tests.js pour tester Effect.Opacity

```
function applyEffect(e) {
    Event.stop(e);
    new Effect.Opacity('byebye', { duration: 2, from: 1, to: 0 });
} // applyEffect

function bindTestElements() {
    Event.observe('byebye', 'click', applyEffect);
} // bindTestElements

Event.observe(window, 'load', bindTestElements);
```

Listing 7-3 Notre feuille de styles, toute simple

```
#byebye {
    font-size: large;
    width: 60ex;
    line-height: 1.5em;
    cursor: default;
}
```

Notez que nous avons, volontairement, ralenti l'effet en calant sa durée sur 2 secondes au lieu d'une. Par ailleurs, puisqu'il s'agissait ici de réduire l'opacité, il a fallu inverser les valeurs par défaut des options `from` et `to`. Voyons le résultat en actionnant le bouton.

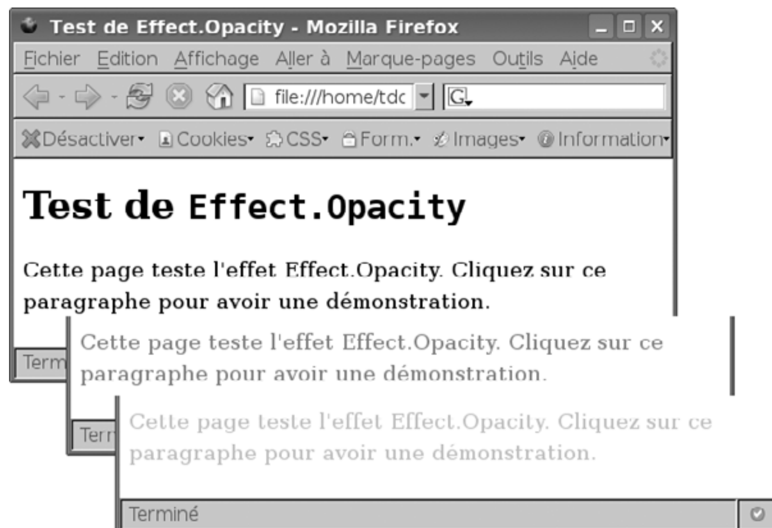


Figure 7–1
L'effet d'opacité en action

À présent, copions notre répertoire sous le nom `scale`, et voyons `Effect.Scale`, qui dispose de nombreuses options. Nous allons brièvement les passer en revue, puis en appliquer quelques-unes.

Tableau 7–4 Les options spécifiques à Effect.Scale

Option	Description
`scaleX`	Active le redimensionnement horizontal (`true` par défaut).
`scaleY`	Active le redimensionnement vertical (`true` par défaut).
`scaleContent`	Active le redimensionnement du contenu (`true` par défaut).
`scaleFromCenter`	Maintient la position du centre au fil du redimensionnement (`false` par défaut).
`scaleMode`	Trois possibilités : - `'box'` (par défaut) : redimensionne la partie visible de l'élément. - `'content'` : redimensionne la totalité de l'élément, en prenant en compte les parties non visibles (qui nécessitaient un défilement, etc.). - objet anonyme avec propriétés `originalWidth` et `originalHeight`, décrivant la taille de départ à utiliser.
`scaleFrom`	Indique le pourcentage de départ à utiliser (par rapport à la taille originale ; `100` par défaut).

Que d'options ! Je vous accorde que la différence entre les modes `box` et `content` est un peu obscure... Notez par ailleurs que la construction exige un paramètre (deuxième argument), qui est le pourcentage final de redimensionnement (ainsi, 50 aura réduit la taille par 2, 110 aura gagné 10 %).

Adaptons donc notre script pour réaliser un agrandissement horizontal centré à 150 %, qui ne touchera pas à la taille du contenu. Après avoir ajusté le HTML pour qu'il indique le bon effet, il suffit de modifier `applyEffect` comme suit :

Listing 7-4 Un exemple de redimensionnement

```
function applyEffect(e) {
    Event.stop(e);
    new Effect.Scale('byebye', 150, {
        duration: 2, scaleY: false, scaleContent: false,
        scaleFromCenter: true
    });
} // applyEffect
```

Et nous allons ajuster la CSS pour que le résultat soit plus parlant.

Listing 7-5 Notre CSS ajustée pour renforcer l'impact visuel de l'effet

```
#byebye {
    margin: 1em auto 0;
```

```
    width: 30ex;
    font-size: 100%;
    line-height: 1.5em;
    text-align: center;
    cursor: default;
    background: #cd9;
}
```

Et voici le résultat !

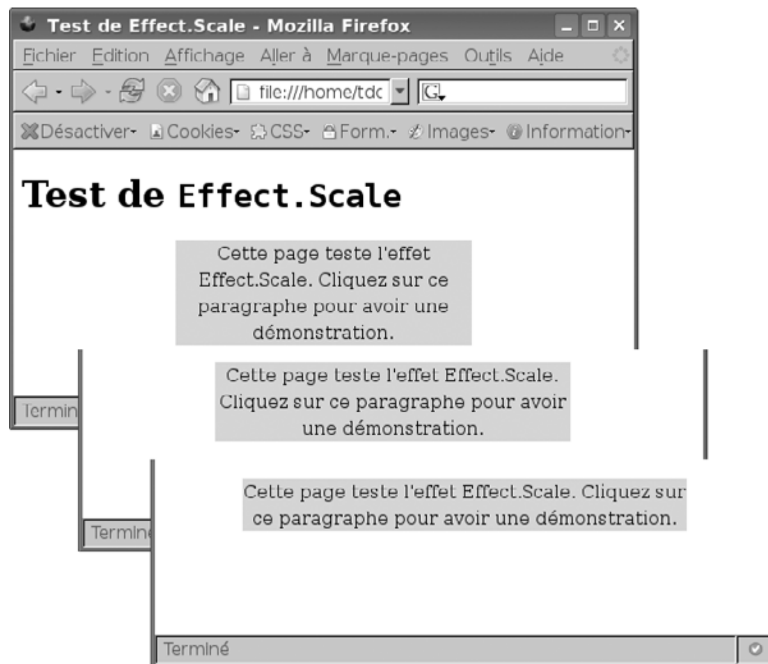


Figure 7–2
L'effet Scale en action

Il faut bien comprendre d'où viennent les dimensions originales du paragraphe : la largeur est fixée par CSS (`30ex`), mais la hauteur est dynamique, simple fonction du nombre de lignes. C'est pourquoi, au fur et à mesure de l'élargissement du paragraphe, les lignes se font suffisamment longues pour être moins nombreuses, et la hauteur diminue (puisque l'effet, lui, n'y touche pas explicitement, conformément à notre option `scaleY: false`).

Cela ne se voit pas forcément à l'impression, mais en testant, vous remarquerez que le centre horizontal du paragraphe est fixe lors du redimensionnement : c'est le rôle de notre option `scaleFromCenter`.

Voyons à présent un exemple de `Highlight`. Cet effet est simple, mais il est très couramment utilisé pour mettre en exergue visuelle un élément de la page qui a changé ou vient d'apparaître (une pratique incontournable pour améliorer l'accessibilité en Ajax). L'effet a trois options spécifiques.

Tableau 7–5 Les options spécifiques à Effect.Highlight

Option	Description
`startcolor`	Expression de couleur CSS indiquant la couleur de fond en début d'effet. Par défaut, `'#ffff99'`, équivalent de `'#ff9'`, soit un jaune clair.
`endcolor`	Même chose mais pour la couleur de fin. Par défaut la couleur de fond appliquée à l'élément (propriété CSS `background-color`), ou à défaut `'#ffffff'`, donc blanc.
`restorecolor`	Définit la couleur de fond après l'effet. Indéfinie par défaut, elle amène script.aculo.us à tenter de récupérer la couleur de fond courante de l'élément (celle avant que l'effet ne démarre), ce qui n'est garanti pour tous les navigateurs que si celle-ci est indéfinie ou exprimée avec la syntaxe `rgb(`*rouge*`, `*vert*`, `*bleu*`)`.

Je vous encourage à systématiquement utiliser le `new` devant le nom de l'effet, mais dans ce cas précis, c'est absolument obligatoire : l'effet ne fonctionnera tout simplement pas sinon.

Copiez votre répertoire précédent dans un nouveau répertoire nommé `highlight`, ajustez à nouveau le HTML pour refléter l'effet, puis modifiez la fonction `applyEffect` du script comme suit.

Listing 7-6 Un exemple de mise en exergue (*highlight*)

```
function applyEffect(e) {
    Event.stop(e);
    new Effect.Highlight('byebye', { duration: 2 });
} // applyEffect
```

La couleur de fond prédéfinie de notre paragraphe nous permet de voir ici la restauration de la couleur de fond originale : on a un fondu du jaune clair vers le vert clair, c'est impeccable !

Une impression noir et blanc ne donnerait rien de bien utile, aussi pour cette fois, on se passera d'une figure...

Enfin, nous allons terminer en beauté avec un exemple un peu plus avancé : la combinaison d'effets synchronisés sur un même élément. On réalise ceci avec `Effect.Parallel`, et de nombreux effets combinés (par exemple `Puff`, `DropOut`, `Grow` et `Shrink`) s'en servent.

Nous allons combiner un changement d'opacité et un rétrécissement, ce qui n'est pas sans rappeler `Effect.Puff`.

Copiez votre répertoire dans un nouveau répertoire `parallel`, ajustez le HTML et modifiez `applyEffect` comme suit.

Listing 7-7 Un exemple d'exécution en parallèle d'effets sur un même élément

```
function applyEffect(e) {
    Event.stop(e);
    new Effect.Parallel([
        new Effect.Opacity('byebye', { sync: true, from: 1, to: 0.33 }),
         new Effect.Scale('byebye', 40,
                            ➥ { sync: true, scaleFromCenter: true })
    ], { duration: 2 });
} // applyEffect
```

Faites bien attention à la syntaxe ! Le constructeur prend deux arguments : un tableau d'effets et les options. Chaque effet doit par ailleurs activer son option `sync`, comme indiqué plus haut. On a ici une réduction d'opacité jusqu'à 33 %, combinée à une réduction de taille jusqu'à 40 %.

L'exécution est impressionnante :

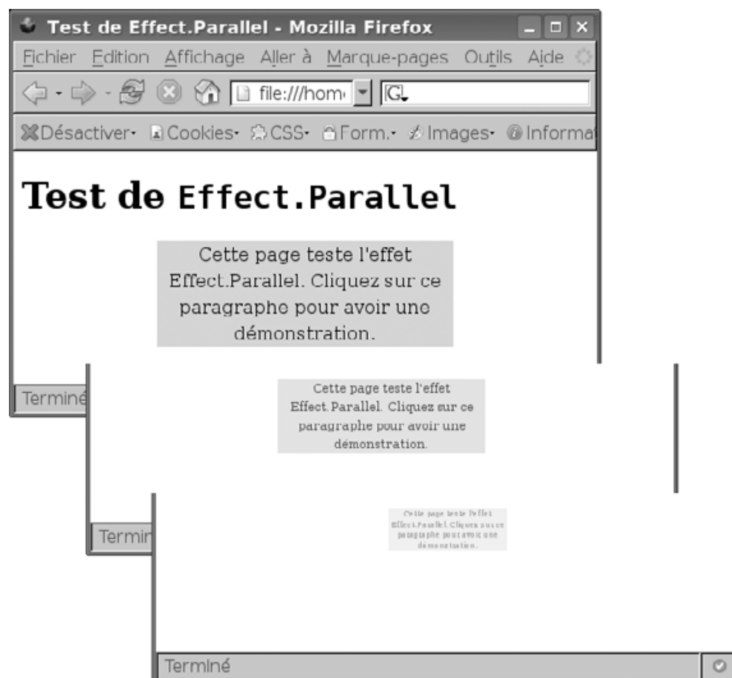


Figure 7–3
Notre effet combiné personnel !

À titre de bonus, voici un exemple encore plus avancé, très proche des effets combinés officiels, qui montre comment enchaîner un masquage de l'élément concerné par ces effets (en supposant qu'ils s'appliquent bien tous au même) une fois l'exécution parallèle terminée, à l'aide d'une fonction de rappel.

Listing 7-8 Ajout d'une fonction de rappel pour masquer finalement l'élément

```
function applyEffect(e) {
    Event.stop(e);
    new Effect.Parallel([
        new Effect.Opacity('byebye', { sync: true, from: 1, to: 0.33 }),
        new Effect.Scale('byebye', 40,
                        ➥ { sync: true, scaleFromCenter: true })
    ], {
        duration: 2,
        afterFinish: function(effect) {
            Element.hide(effect.effects[0].element);
        }
    });
} // applyEffect
```

Souvenez-vous : les fonctions de rappel reçoivent l'effet conteneur (donc notre `Effect.Parallel`), lequel recense ses effets synchronisés dans une propriété tableau `effects`. Puisque nous savons ici que tous les effets synchronisés référencent le même élément, on prend le premier, et on passe sa propriété `element` à la fonction Prototype `Element.hide`. Et voilà !

Les effets combinés

Les effets combinés sont constitués soit d'effets noyau spécialement paramétrés, soit d'exécutions parallèles de tels effets (réalisées avec `Effect.Parallel`), soit de séquences de tels effets (réalisées à l'aide de fonctions de rappel).

Attention, certains effets n'auront pas forcément le résultat désiré sur votre navigateur. Les deux principales causes sont l'absence de prise en charge de la propriété CSS `opacity` (affecte `Appear`, `Fade` et `Pulsate`), par exemple sur Konqueror (à l'heure où j'écris ces lignes, en version 3.5.2) et la configuration d'une taille minimale pour les polices de caractères, qui va empêcher la réduction extrême de textes (ce qui affecte notamment `Shrink` et `Scale`). Soyez avertis !

À présent, dressons une liste rapide des effets combinés. Au passage, notez que vous trouverez une page faisant la démonstration de tous ces effets ici :

http://wiki.script.aculo.us/scriptaculous/show/CombinationEffectsDemo.

Tableau 7–6 Les effets combinés de script.aculo.us

Effet	Description
`Effect.Appear`	Affiche l'élément en supprimant l'éventuel `display: none` de son attribut `style`, et en faisant graduellement passer son opacité de 0 % à 100 %. Hormis l'aspect `display`, équivalent à un `Opacity`.
`Effect.Fade`	Symétrique de `Appear` : ramène l'opacité vers 0 % et ajoute un `display: none` au style.
`Effect.Puff`	Donne l'illusion que l'élément part en fumée (mélange d'un `Fade` et d'un `Grow` en direction `center`) ! On préférera l'appliquer sur des éléments positionnés (en absolu ou relatif), pour éviter les décalages subis à gauche au déclenchement de l'effet.
`Effect.BlindDown`	Affiche l'élément de haut en bas, sans modifier sa position dans la page ou sa présence.
`Effect.BlindUp`	Symétrique de `BlindDown`, mais de bas en haut.
`Effect.SwitchOff`	Simule l'extinction d'une ancienne télévision : un clignotement suivi d'un ramassement vers la ligne centrale. L'élément disparaît en fin d'effet.
`Effect.SlideDown`	Similaire à `BlindDown`, si ce n'est que le contenu de l'élément suit le glissement, comme s'il était fixé sur un volet qu'on baissait. A tendance à clignoter quand le contenu est riche...
`Effect.SlideUp`	Symétrique à `SlideDown`, mais de bas en haut. Même remarque.
`Effect.DropOut`	En plus d'un `Fade`, l'élément disparaît en « tombant ». Il part en quelque sorte aux oubliettes.
`Effect.Shake`	Déplace l'élément alternativement de droite à gauche.
`Effect.Pulsate`	Fait « pulser » l'élément, en 5 cycles `Appear`/`Fade` (sans finir par `display: none`, en revanche).
`Effect.Squish`	Écrase l'élément vers son coin supérieur gauche, et le fait finalement disparaître.
`Effect.Fold`	Similaire à `Squish`, mais réduit d'abord la hauteur, puis la largeur.
`Effect.Grow`	Amène l'élément d'une taille nulle à sa taille courante, en suivant une option `direction` de grossissement (`top-left`, `top-right`, `bottom-left`, `bottom-right`, ou la valeur par défaut : `center`).
`Effect.Shrink`	Inverse de `Grow` : réduit l'élément jusqu'à disparition. Même option spécifique.
`Effect.toggle`	Il ne s'agit pas tant d'un effet que d'une méthode de bascule d'état par effets (voir section suivante).

Effect.toggle

Cette fonction permet de faire basculer l'état d'un élément, de visible à invisible et inversement. Elle a un premier argument obligatoire, qui est bien sûr l'élément (ou son ID). Le deuxième argument est optionnel, et indique la famille d'effets à utiliser pour assurer la transition d'un état à l'autre. Trois valeurs sont possibles :

- `'appear'`, valeur par défaut, utilisera `Appear` et `Fade`.
- `'blind'` utilisera `BlindDown` et `BlindUp`.
- `'slide'` utilisera `SlideDown` et `SlideUp`.

De cette façon, nous n'avons pas à tester manuellement l'état affiché (propriété `display`) de l'élément dans nos scripts. Notez qu'il n'est pas nécessaire d'utiliser la même famille tout du long : un `Effect.toggle(element, 'blind')` affichera sans problème un élément préalablement caché par un `Effect.toggle(element)`, par exemple.

Quelques précisions importantes

Avant que vous ne vous jetiez sur vos tests personnels, permettez-moi de résumer les principaux points techniques à surveiller en manipulant ces effets :

- On ne le répétera jamais assez : pour masquer un élément d'entrée de jeu, il faut enfreindre légèrement la séparation du contenu et de la forme en définissant une propriété `display: none` dans son attribut `style`, et non dans une règle CSS. Sans quoi, script.aculo.us ne pourra pas l'afficher à nouveau.
- Pour les effets combinés comme pour les effets noyau, assurez-vous toujours d'utiliser `new` devant le nom de l'effet, afin d'éviter les surprises fâcheuses.
- La plupart des effets fonctionnent principalement sur des éléments de type bloc (voir l'annexe B pour plus de détails sur cette notion), à l'exception des éléments relatifs aux tables (par exemple `table`, `tr`, `td`, `thead`, `caption`), dont les valeurs pour la propriété `display` sont particulières et d'une gestion complexe.
- `SlideDown` et `SlideUp` ont quelques exigences sur l'aspect de votre balisage, et des besoins supplémentaires si vous les appliquez à des éléments en positionnement absolu dans MSIE. Voyez la documentation pour les détails.

Vous trouverez une documentation pour ces effets sur le wiki documentaire de script.aculo.us : http://wiki.script.aculo.us/scriptaculous/show/CombinationEffects.

Des commentaires qui font de l'effet

À titre de démonstration, nous allons reprendre notre dernier exemple de saisie de commentaires, que nous avons adapté au début de ce chapitre, pour y ajouter une petite dose d'effets qui ressemblera au comportement d'outils de blog récents, par exemple Typo.

Il va nous falloir procéder avec quelques précautions car nous allons changer de version de Prototype. Commencez par recopier votre répertoire `ajax_updater` dans un répertoire `fancy_comments`. Copiez ensuite dans le sous-répertoire `docroot` les fichiers `prototype.js`, `scriptaculous.js` et `effects.js` d'un des répertoires de test d'effet, par exemple `parallel`.

Commençons par ajuster notre HTML en ajoutant dans le `head` le chargement de script.aculo.us, et en ajoutant un fragment pour l'affichage du nombre de commentaires.

Listing 7-9 Chargement de script.aculo.us et affichage du nombre de commentaires

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Des commentaires qui font de l'effet !</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="scriptaculous.js?load=effects">
    </script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

...

</form>

<p>Il y a <span id="commentCount">0 commentaire</span>.</p>

<div id="comments"></div>

</body>
</html>
```

Voici les modifications que nous souhaitons apporter :

- Nous affichons désormais le nombre de commentaires, comme vous avez pu le voir dans le listing ci-dessus. À chaque ajout, ce nombre clignotera, pour attirer l'attention sur la modification.
- Le nouveau commentaire apparaîtra parallèlement à l'aide d'un effet `BlindDown` (qui a l'avantage, par rapport à `SlideDown`, de ne rien exiger de particulier quant au XHTML de l'élément). En revanche, afin d'éviter un clignotement dû à son bref affichage suite à l'insertion dans le DOM, on va modifier notre modèle, `commentaire.rhtml`, pour utiliser un attribut `style` masquant d'entrée de jeu le bloc du commentaire.

Voici le nouveau modèle.

Listing 7-10 La version ajustée (extrait ciblé) du modèle de commentaire

```
<div class="comment" style="display: none;">
…
</div>
```

Et la nouvelle version de notre fonction `switchToAJAX` !

Listing 7-11 L'ajout de commentaire, façon script.aculo.us

```
function switchToAJAX(e) {
    …
    new Ajax.Updater('comments', '/add_comment',
    {
        postBody: data,
        insertion: Insertion.Bottom,
        onComplete: function() {
            var comments = $$('#comments .comment');
            var commentCount = comments.length;
            var lastComment = comments.last();
            var text = commentCount + ' commentaire';
            if (commentCount > 1)
                text += 's';
            Element.update('commentCount', text);
            new Effect.Parallel([
                new Effect.Pulsate('commentCount', { sync: true }),
                new Effect.BlindDown(lastComment, { sync: true })
            ], { duration: 2 });
        }
    });
} // switchToAJAX
```

Comme quoi, il y a des cas où `onComplete` est utile pour un `Ajax.Updater`... Notez ici la fonction `$$` de Prototype 1.5, qui permet d'interpréter un sélecteur CSS et renvoie un tableau des éléments correspondants. Même si, à l'heure actuelle, le sélecteur > (élément fils) n'est pas encore reconnu, cela reste très utile pour récupérer dynamiquement notre nombre de commentaires ! Si la syntaxe de notre sélecteur CSS vous intrigue, faites donc un tour du côté de l'annexe B.

En espérant que cela donne quelque chose une fois imprimé (et dans le cas contraire, qu'attendez-vous pour tester ?), la figure 7-4 vous donne un aperçu du résultat :

Files d'effets

Par défaut, lorsqu'on crée plusieurs effets, il s'exécutent en parallèle. Vous allez sûrement vous étonner, alors, qu'il existe un `Effect.Parallel`. Celui-ci est pourtant très utile, puisqu'il assure que les effets qu'il enrobe seront synchronisés : mêmes moments de démarrage et de fin d'exécution. Lancer plusieurs effets sans passer par lui les démarre un par un et les exécute à leur rythme : s'ils s'exécutent en parallèle, ils ne sont pas pour autant synchrones. Donc oui, c'est vrai, `Effect.Parallel` devrait plutôt se nommer `Effect.Synchronized`.



Figure 7–4
L'ajout de commentaire avec effets

L'exécution simultanée est parfois problématique

Reste que cette exécution parallèle, qui est active par défaut, peut poser problème : certains effets vont entrer en conflit, et probablement aboutir à un aspect « cassé » du pauvre élément victime. Imaginez par exemple le code suivant :

```
new Effect.BlindUp('accountInfo');
new Effect.BlindDown('accountInfo');
```

Là où vous vouliez visiblement enchaîner les effets, leur exécution simultanée donne un résultat catastrophique (clignotement et, le plus souvent, élément invisible ou partiellement visible au bout du compte).

Les files, ou comment enchaîner les effets

Les files sont justement là pour permettre d'ordonner les effets. Dans script.aculo.us, on trouve un référentiel global de files nommé `Effect.Queues`, qui se comporte comme un tableau associatif d'objets `Effect.ScopedQueue`. Chaque file est nommée. Par défaut, il en existe une seule, nommée `'global'`, qui est d'ailleurs accessible directement via la référence globale `Effect.Queue`. C'est dans cette file globale que vos effets sont ajoutés par défaut.

Quand un effet est ajouté à une file, il faut préciser notamment si on souhaite l'exécuter classiquement (sans planification particulière), ou le placer en début ou en fin de file. Ce détail est fourni par une option commune à tous les effets, que nous avons déjà évoquée : `queue`. Lorsque cette option manque, l'effet est ajouté pour exécution simultanée dans la file globale. Sinon, elle peut valoir `'front'` ou `'end'` (les noms parlent d'eux-mêmes), ou être un objet anonyme précisant une file particulière (nous verrons cela dans un moment).

Ce qu'il faut bien comprendre, c'est le moment auquel un effet est exécuté : dès qu'une file a au moins un effet, elle va examiner régulièrement (dans la version actuelle, toutes les 40 ms, soit 25 fois par seconde, ce qui est largement suffisant) ses effets pour lancer ceux dont le moment de démarrage est arrivé, faire évoluer ceux dont le démarrage est passé, et retirer ceux qui auront terminé. Cette notion de moment de démarrage (propriété `startOn` de l'effet, déjà évoquée) est donc critique.

Par défaut, un effet démarre immédiatement (dès son premier examen par la file, au pire 40 ms après sa définition), ou après une attente de `delay` millisecondes, s'il est pourvu d'une option `delay`.

Mais l'utilisation d'une position explicite d'ajout dans la file modifie les moments de démarrage des effets de la file :

- pas d'information de position : ajout sans modifier quelque moment d'exécution que ce soit.
- `'front'` : ajout en début de file, et décalage des démarrages de tous les effets suivants (les effets qui avaient démarré très récemment risquent d'être suspendus !).
- `'end'` : ajout en fin de file, le moment de démarrage de l'effet étant ajusté pour pouvoir exécuter tous les autres avant.

Il faut se méfier de files trop longues, par exemple en définissant une dizaine d'effets successivement pour la même file (ce qui serait probablement bien trop lourd visuellement en termes d'ergonomie, de toutes façons). En effet, il ne faut pas oublier qu'au bout de 40 ms maximum, la file va commencer à traiter ses effets. Ajouter un effet en position `'front'` plus de 40 ms après l'insertion du premier effet dans la file peut donner des résultats inattendus... Si vous avez besoin de définir plus de 2 ou 3 effets dans une même séquence, prenez soin de les définir dans le bon ordre, en position `'end'`.

Exemple d'enchaînement

Imaginons la séquence de définitions suivantes, en supposant que son exécution prendra moins de 40 ms :

```
new Effect.Highlight('userPad', { duration: 1.5 });
new Effect.Scale('userPad', { queue: 'end' });
new Effect.Appear('userPad', { queue: 'front' });
```

On obtient la file suivante :

1 `Appear` prévu pour exécution immédiate, durée par défaut 1s.
2 `Highlight` prévu pour exécution à 1 s (après la fin de `Appear`).
3 `Scale` prévu pour exécution à 2,5 s (après la fin de `Highlight`, de durée 1,5 s).

Utiliser plusieurs files

Ce n'est déjà pas mal, mais cela ne suffit pas toujours. En effet, on peut avoir plusieurs séquences d'effets, pour plusieurs portions de la page. Prenons un exemple, un peu forcé il est vrai : sur un site marchand, suite à l'ajout d'un achat dans le panier depuis une page de détails et au retour sur le catalogue, on pourrait avoir la séquence suivante :

1 Affichage et `Highlight` d'un message de confirmation en haut de page.
2 Trois secondes après la fin du `Highlight`, `Fade` de la confirmation.

Mais s'il s'agit du premier achat, on pourrait vouloir exécuter, parallèlement, la séquence suivante :

1 `BlindDown` du résumé de panier dans la barre latérale.
2 Une fois le `BlindDown` terminé, `Appear` du bouton de validation de commande sous le résumé.

Tout ceci a l'air attrayant, mais comment nous y prendre pour avoir deux files distinctes ? Rien de plus simple : il suffit d'en nommer au moins une. Pour pouvoir nommer la file à laquelle l'effet doit s'ajouter, il faut changer la nature de notre argument `queue` : au lieu d'une `String`, il va s'agir d'un objet anonyme avec une propriété `scope` (le nom de la file) et, si besoin est, une propriété position (toujours `'front'` ou `'end'`).

Voici l'implémentation des besoins décrits ci-dessus.

Listing 7-12 Une définition avancée : deux files d'effets

```
// 1ère file
new Effect.Highlight('notice', { queue: { scope: 'notice' } });
new Effect.Fade('notice', {
    delay: 3, queue: { scope: 'notice', position: 'end' }
});
```

```
// 2eme file
new Effect.BlindDown('cart', { queue: { scope: 'cart' } });
new Effect.Appear('btnOrder', {
    queue: { scope: 'cart', position: 'end' }
});
```

Et voilà, ce n'était pas si compliqué !

Pour conclure, parlons de la surcharge de file. Lorsqu'un utilisateur sollicite trop l'interface (en cliquant à répétition sur un bouton, par exemple), on peut aboutir à un tel enchaînement d'effets qu'ils continuent inutilement leur exécution plusieurs secondes après que l'utilisateur... se soit calmé, dirons-nous. C'est souvent inutile.

Lorsque cela a du sens ergonomiquement, vous veillerez donc à poser une limite au nombre d'effets qui peuvent déjà se trouver dans la file avant d'insérer celui que vous êtes en train de définir. C'est tout simple : il suffit d'ajouter une propriété `limit` à la définition de file, par exemple :

```
new Effect.BlindDown('companyInfo', {
    queue: { scope: 'company', position: 'end', limit: 2 }
});
```

Glisser-déplacer

À présent que nous avons fait le tour des effets, il est temps de se pencher sur une autre fonctionnalité majeure de script.aculo.us : le glisser-déplacer. Ceux d'entre vous qui ont déjà tenté d'écrire leur propre gestion de glisser-déplacer en JavaScript portable ont goûté l'enfer. Avec script.aculo.us, plus de souci, une gestion solide est enfin disponible !

La gestion du glisser-déplacer dans script.aculo.us repose sur deux classes, que nous allons voir séparément : `Draggable` et `Droppables`. Ces classes sont fournies par le module `dragdrop.js`, qu'il faudra charger. Le plus simple pour ce faire est de modifier notre appel à script.aculo.us :

```
<script type="text/javascript"
        ➥ src="scriptaculous.js?load=effets,dragdrop"></script>
```

Commençons par examiner les possibilités de glissement.

Faire glisser un élément avec Draggable

Pour pouvoir faire glisser un élément, il suffit de créer un objet `Draggable` basé sur l'élément, au chargement de la page. Le listing ci-après propose un exemple.

Listing 7-13 Altération d'un élément au chargement pour qu'il puisse glisser

```
function initDraggables() {
    new Draggable('myWizzyDiv');
} // initDraggables

Event.observe(window, 'load', initDraggables, false);
```

Comme vous pouvez le voir, c'est extrêmement simple. Précisons toutefois qu'on n'utilisera pas comme éléments des champs de formulaire, en tout cas pas directement (on utilisera par exemple leur paragraphe conteneur), en raison de problèmes sur certains navigateurs.

C'est donc simple, mais si la fonctionnalité s'arrêtait là, nous buterions rapidement sur ses limitations. On pourrait par exemple vouloir :

- limiter la zone au sein de laquelle l'élément peut être déplacé ;
- demander à l'élément de se déplacer par paliers (ce qu'on appelle un *snap*) d'un certain nombre de pixels plutôt que pixel par pixel ;
- demander à l'élément de revenir à sa position initiale une fois lâché ;
- altérer les effets visuels déclenchés à la prise et au relâchement ;
- modifier la position Z de l'élément pour qu'il soit masqué par certains autres (rarement utile, ceci dit) ;
- préférer limiter la zone de prise à un élément donné plutôt qu'à toute la zone de l'élément concerné (pour réaliser une poignée, par exemple).

On le voit, les exigences concrètes peuvent être très variées sur de véritables projets. Heureusement pour nous, `Draggable` est à même de répondre à tous ces besoins (et même plus) ! Il utilise pour cela des options, passées, de façon classique, comme propriétés d'un objet anonyme fourni en deuxième argument. Les propriétés disponibles sont les suivantes.

Tableau 7–7 Propriétés prises en charge par Draggable

Option	Description
`zindex`	Position Z de l'élément (1 000 par défaut, ce qui assure a priori que l'élément est toujours visible).
`revert`	Demande à l'élément d'exécuter `reverteffect` une fois relâché (`false` par défaut). On peut aussi simplifier en précisant directement la fonction à invoquer, et laisser `reverteffect` tranquille.
`constraint`	Peut valoir `'vertical'` ou `'horizontal'`, ce qui limite la direction autorisée pour le glissement. Indéfini par défaut.
`delay`	Temps en millisecondes pendant lequel le bouton de la souris doit être enfoncé avant que le glisser-déplacer ne puisse avoir lieu. Vaut zéro (désactivé) par défaut.

Tableau 7–7 Propriétés prises en charge par Draggable (suite)

Option	Description
`snap`	Gère l'ajustement de la position au fil du glissement. Peut prendre de nombreuses formes : – `false` (valeur par défaut) : pas d'ajustement ou de limitation. – nombre entier : taille du *snap* en pixels, par exemple `10` pour se déplacer par paliers de 10 pixels. – Tableau de deux nombres entiers : différenciation horizontale et verticale, par exemple `[10, 20]` utilisera un palier horizontal de 10 et un palier vertical de 20. – Fonction : prend la position prévue en arguments (X, Y) et renvoie un tableau avec la position ajustée `[ax, ay]`. Permet d'assurer un *snap* uniquement à certains endroits, par exemple au bord de certains éléments, et de limiter la zone de glissement autorisée !
`handle`	Permet de définir un élément servant de poignée (*handle*) pour le glissement. Par défaut vaut `null`, de sorte que toute la surface de l'élément est utilisable pour démarrer le glissement. Peut prendre plusieurs autres valeurs : – nom de classe CSS : le premier élément fils ayant cette classe servira de poignée[a] ; – ID d'élément ou élément directement : désigne l'élément devant servir de poignée. Une utilisation classique consiste à limiter le déclenchement du glissement à une «barre de titre» pour un élément, généralement représentée sous forme d'un élément `div` ou de titre (`h`*x*) fils.
`ghosting`	Si actif, déplace un clone de l'objet plutôt que l'objet lui-même, jusqu'au dépôt. Vaut `false` par défaut.
`starteffect`	Fonction appelée au démarrage du glissement. Par défaut, sauvegarde l'opacité actuelle de l'élément puis l'amène à 70 % à l'aide d'un `Effect.Opacity`. Reçoit l'élément en argument.
`endeffect`	Fonction appelée au relâchement. Par défaut, restaure l'opacité sauvegardée (100 % si elle n'a pas été déterminée au démarrage) à l'aide d'un `Effect.Opacity`. Reçoit l'élément en argument.
`reverteffect`	Fonction appelée au relâchement, après `endeffect`, si la propriété `revert` est à `true`. Par défaut, ramène l'élément à sa position d'origine, en un temps proportionnel à la distance de glissement, à l'aide d'un `Effect.Move`. Reçoit trois arguments : l'élément et les composantes verticale et horizontale du glissement.
`scroll`	Indique si le glissement doit, lorsqu'il atteint la limite d'affichage d'un élément conteneur, déclencher le défilement de cet affichage pour pouvoir continuer à glisser. Vaut `false` par défaut, mais peut référencer un objet conteneur par son ID ou directement, par exemple l'objet global `window`. Du coup, en atteignant le bord de la page, celle-ci va se mettre à défiler (même si elle n'en avait pas besoin, car tout son contenu était visible !) pour permettre de prolonger le défilement.
`scrollSensitivity`	Niveau de proximité aux bords du conteneur pour déclenchement du défilement. En pixels ; vaut `20` par défaut.
`scrollSpeed`	Vitesse de défilement, en pixels par progression du glissement. Vaut `15` par défaut.

a. Cette possibilité semble défectueuse dans la version 1.6.2...

Notez que les propriétés de gestion du défilement sont encore assez expérimentales, et ne fonctionnent pas forcément partout. Ceci dit, elles correspondent à un type d'utilisation assez limité.

Eh bien, que d'options ! Réalisons un petit exemple pour nous y retrouver.

Commencez par copier votre répertoire `opacity` dans un nouveau répertoire `draggable`. Ajoutez-y le fichier `dragdrop.js` de la bibliothèque. Nous allons ensuite modifier la page HTML pour préparer notre exemple.

Listing 7-14 Notre page HTML avec un pavé déplaçable

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Test de Draggable</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,dragdrop"></script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test de <code>Draggable</code></h1>

<div id="wizzy">
    <div id="wizzy-menubar"></div>
    <div class="contents">Faites-moi glisser !</div>
</div>

</body>
</html>
```

La feuille de styles, `tests.css`, est assez simple.

Listing 7-15 La feuille de styles pour notre exemple

```
#wizzy {
    width: 25ex;
    border: 1px solid navy;
    background: #ddf;
    color: blue;
    text-align: center;
    font-size: large;
```

```
}

#wizzy-menubar {
    height: 0.5em;
    background: navy;
    cursor: move; /* indice visuel pratique */
}

.contents {
    padding: 1em;
}
```

Nous obtenons l'aspect suivant.



Figure 7–5 Notre page avant qu'on fasse glisser le bloc

Notez la partie haute du pavé, sa « barre de titre » en quelque sorte, qui va servir de seule zone possible pour déclencher le glissement, comme on le voit dans le script ci-dessous, `tests.js` :

Listing 7-16 Notre script de tests

```
MAX = 400;

function initDraggables() {
    new Draggable('wizzy', {
        revert: true, handle: 'wizzy-menubar',
        snap: function(x, y) {
            return [ [ x, MAX ].min(), [ y, MAX ].min() ];
        }
    });
} // initDraggables

Event.observe(window, 'load', initDraggables);
```

Pour cet exemple, nous utilisons :

- `revert`, pour demander à l'élément de revenir à sa position initiale une fois relâché.
- `handle`, pour préciser l'ID de l'élément qui servira de zone de déclenchement. Cliquer sur le corps du pavé ne servira donc à rien, comme pour la plupart des systèmes de fenêtrage par exemple.
- `snap`, d'une façon avancée : nous définissons notre propre fonction de contrôle, qui limitera les déplacements à la position (400, 400), donc des limites droite et basse.

Le pavé bénéficie des comportements par défaut pour les effets de démarrage et d'arrêt (ajustement de l'opacité à 70 %), ainsi que pour le relâchement (retour à la position de départ).

Chargez la page et amusez-vous à déplacer le bloc !



Figure 7–6 Notre pavé en cours de déplacement

Désactiver la possibilité de faire glisser un élément

Si vous souhaitez empêcher, temporairement ou définitivement, un élément de glisser, il suffit de conserver une référence sur le `Draggable` que vous créez pour lui, et de la passer par la suite à `Draggables.unregister` (attention au « s » : c'est un pluriel). Vous pourrez réactiver le glissement plus tard en appelant `Draggables.register` (ce qui avait été fait automatiquement quand vous aviez créé le `Draggable`).

Réagir aux étapes des glissements

Il est possible d'inscrire des observateurs auprès de `Draggables`, à l'aide de sa méthode `addObserver` (et bien sûr, on peut se désinscrire avec `removeObserver`). Chaque observateur peut implémenter jusqu'à trois méthodes correspondant aux trois catégo-

ries d'événement dans un glissement : `onStart`, `onDrag` et `onEnd`. Seule `onDrag` peut être appelée plusieurs fois (chaque fois que le glissement évolue dans l'espace).

Les trois méthodes acceptent trois arguments :

1. Le nom de l'événement (`'onStart'`, `'onDrag'` ou `'onEnd'`), ce qui permet par exemple d'utiliser une même méthode pour plusieurs événements.
2. L'objet `Draggable` concerné (puisqu'on enregistre les observateurs au niveau global, avec `Draggables.addObserver`).
3. L'objet événement, manipulable avec le `Event` de Prototype.

Voici un exemple simple, utilisant l'objet global `console` de Firebug :

```
Draggables.addObserver({
    onStart: function(eventName, draggable) {
        console.log(draggable.id + ' : ' + eventName);
    },
    onEnd: onStart
});
```

Gérer le dépôt d'un élément avec Droppables

Faire glisser, c'est bien, mais si c'est pour ne pas déposer dans un endroit précis, ça ne sert qu'à réarranger la page (ce qui n'est déjà pas si mal).

Il est bien entendu possible, avec script.aculo.us, de définir des éléments de la page comme étant des zones de dépôt (ce que la bibliothèque appelle des *droppables*).

Chaque zone peut préciser certaines conditions d'acceptation pour un dépôt (par exemple, « uniquement les éléments ayant la classe X » ou « uniquement ceux issus du conteneur Y »), réagir au survol d'un candidat valide en cours de déplacement, et bien sûr, réagir au dépôt à proprement parler.

Il ne s'agit pas ici de créer un objet d'enrobage, mais simplement d'inscrire ou de désinscrire notre élément dans le référentiel global. Là où on avait `Draggable`, `Draggables.register` et `Draggables.unregister`, on a ici juste `Droppables.add` et `Droppables.remove`.

L'ajout prend en charge plusieurs options. Toutes, sauf `onDrop`, sont optionnelles.

Tableau 7–8 Options prises en charge pour l'inscription d'une zone de dépôt

Option	Description
`accept`	Nom ou tableau de noms de classes CSS, qui établit un filtre : seuls les éléments disposant d'au moins une des classes spécifiées auront le droit d'être déposés.
`containment`	Référence ou tableau de références d'éléments (ID ou références directes), qui établit un filtre lui aussi : seuls les éléments contenus dans un des conteneurs spécifiés auront le droit d'être déposés. Cumulatif avec `accept`.

Tableau 7–8 Options prises en charge pour l'inscription d'une zone de dépôt (suite)

Option	Description
hoverclass	Classe CSS ajoutée à la zone de dépôt lorsqu'un élément acceptable est en train de glisser dessus. Pratique pour indiquer que le dépôt est autorisé.
onDrop	Seule option obligatoire, doit fournir une fonction de gestion du dépôt, qui va probablement synchroniser la couche serveur via Ajax et récupérer un ajustement de la page en réponse.

Il existe en réalité quelques options supplémentaires, mais qui sont principalement utilisées en interne par `Sortables`, le mécanisme de tri dynamique d'éléments par glisser-déplacer que nous verrons à la prochaine section.

Un exemple sympathique de glisser-déposer

À titre d'exemple, nous allons réaliser un équivalent de la démonstration de panier présente sur le site de script.aculo.us, avec une couche serveur qui composera à chaque manipulation le contenu de la zone « panier » (de là à ajouter l'information dans une gestion de sessions, il n'y aurait qu'un pas).

Le principe est simple : nous avons quelques éléments à vendre dans notre magasin, à savoir des tasses et des tee-shirts (le parfait magasin en ligne pour *geeks*). Pour placer un élément dans le panier, nous allons simplement le faire glisser. Ajouter plusieurs fois le même produit fera l'objet d'une gestion de quantité par type de produit. Pour retirer un produit, il suffira de le faire glisser vers une poubelle à côté du panier.

Copiez donc votre répertoire `draggable` dans un nouveau répertoire `fancy_cart/docroot`. Vous pourrez trouver les images de cet exemple dans l'archive des codes source disponible sur le site des éditions Eyrolles, et les déposer dans le répertoire. Commençons par modifier notre HTML.

Listing 7-17 La page HTML pour notre gestion de panier

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Un sympathique panier</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,dragdrop"></script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>
```

```
<h1>Un sympathique panier</h1>

<div id="products">
    <img src="product1.png" id="product_1" alt="Mug" class="product" />
    <img src="product2.png" id="product_2" alt="T-shirt" class="product" />
</div>

<h2>Votre panier :</h2>

<div id="cart" class="cart">
    <div id="items">Votre panier est vide.</div>
    <div id="cartFooter">
        <p id="wastebin"><img src="wastebin.png" alt="Poubelle" /></p>
        <p id="indicator" style="display: none">
            <img src="spinner.gif" />
            Mise à jour&#8230;
        </p>
    </div>
</div>

</body>
</html>
```

La feuille de styles est plus longue que d'habitude mais reste assez simple (et comme toujours, dans le doute, commencez par l'annexe B).

Listing 7-18 La feuille de styles pour notre gestion de panier

```
body {
    background: white;
}

#products {
    margin-bottom: 20px;
    height: 120px;
}

#cart {
    margin-top: 10px;
}

#cartFooter {
    position: relative;
}

#indicator {
    position: absolute;
    left: 90px;
```

```
    top: 0;
    height: 16px;
    font-family: sans-serif;
    font-size: small;
    color: gray;
}

#indicator img {
    vertical-align: middle;
}

#items {
    width: 500px;
    height: 96px;
    border: 1px solid orange;
    padding: 1ex;
    font-family: sans-serif;
    color: gray;
}

#items.active {
    background: #fec;
}

#wastebin {
    margin: 0;
    width: 64px;
    padding: 1ex;
    border: 2px dotted white; /* MSIE ignore 'transparent' */
}

#wastebin.active {
    border-color: #88f;
}

.cartItemsLine {
    font-size: small;
}

.cartItemsLine img {
    height: 32px;
    vertical-align: middle;
}

.product {
    cursor: move;
}
```

Voici à présent l'un des deux gros morceaux de l'exemple : le script côté client.

Listing 7-19 Notre script client pour ce sympathique panier

```
function initDraggables() { ❶
    new Draggable('product_1', { revert: true });
    new Draggable('product_2', { revert: true });
    Droppables.add('items', { ❷
        accept: 'product', hoverclass: 'active',
        onDrop: function(product) { updateCart('add', product); }
    });
    Droppables.add('wastebin', {
        accept: 'cartItem', hoverclass: 'active',
        onDrop: function(product) {
            Element.hide(product);
            updateCart('remove', product);
        }
    });
} // initDragDrop

function toggleIndicator() { ❸
    Element.toggle('indicator');
} // toggleIndicator

function updateCart(mode, product) { ❹
    var id = product.id.split('_')[1];
    new Ajax.Updater('items', '/' + mode, {
        postBody: $H({ 'id': id }).toQueryString(),
        evalScripts: true,
        onLoading: toggleIndicator,
        onComplete: toggleIndicator
    });
} // updateCart

Event.observe(window, 'load', initDragDrop);
```

❶ On commence par déclarer que nos deux produits peuvent être déplacés, et doivent se repositionner une fois lâchés.

❷ On inscrit l’afficheur du panier comme zone de dépôt réservée aux produits, et la poubelle comme zone de dépôt réservée aux éléments du panier (vous verrez que le XHTML renvoyé par le serveur accole une classe `cartItem` aux éléments qui peuvent être déplacés). Pour le coup, il était utile de factoriser l’invocation Ajax...

❸ Simple bascule d’affichage pour notre indicateur (rappel : son `display: none` doit être *inline*, pas dans la CSS !).

❹ L’invocation Ajax, assez avancée ! On appelle soit `/add` soit `/remove`, avec toujours un argument `id` qui identifie le produit, le tout en `post` par défaut. L’appel est encadré par les bascules de visibilité de l’indicateur. Comme on va renvoyer du XHTML contenant du JavaScript, il ne faut pas oublier d’activer `evalScripts` !

Remarquez au passage que, pour tant de traitements, ce script est vraiment court ! Prototype et script.aculo.us sont de bonnes bibliothèques...

Et bien sûr, il reste à écrire la gestion côté serveur. Nous déléguerons la création du fragment XHTML représentant l'intérieur du panier à un modèle interprété par ERb, comme nous l'avons déjà fait plusieurs fois. Voici d'abord le code serveur, dans un fichier `serveur.rb` au-dessus de `docroot`.

Listing 7-20 Le script serveur de gestion du panier, serveur.rb

```
#! /usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

PRODUCT_LABELS = { ❶
    '1' => 'Mug',
    '2' => 'T-shirt'
}

cart = {}

template_text = File.read('cart.rhtml')
cart_html = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/add') do |request, response| ❷
  product_id = CGI::parse(request.body)['id'][0]
  cart[product_id] = cart.include?(product_id) ? cart[product_id ] + 1 : 1
  sleep 1 # Simuler un A/R web, qu'on voie l'indicateur...
  response['Content-Type'] = 'text/html'
  response.body = cart_html.result(binding)
end

server.mount_proc('/remove') do |request, response| ❸
  product_id = CGI::parse(request.body)['id'][0]
  if cart[product_id] > 1
    cart[product_id] = cart[product_id] -1
  else
    cart.delete(product_id)
  end
  sleep 1
  response['Content-Type'] = 'text/html'
  response.body = cart_html.result(binding)
end
```

```
trap('INT') { server.shutdown }

server.start
```

❶ Évidemment, dans une véritable application, on irait pêcher ces noms dynamiquement dans la base de données en réponse aux actions `add` et `remove`...
Notez également l'initialisation d'un unique panier indépendant du client, là où on utiliserait des cookies de session en temps normal.

❷ Ajout de produit : on récupère le paramètre `id` transmis en `post`, puis si on a déjà cet ID dans le panier, on augmente la quantité, sinon on le crée avec une quantité de un.

❸ Suppression de produit : on récupère l'ID, puis s'il y en a plus d'un dans le panier, on diminue la quantité, sinon on retire carrément le produit du panier.

Et voici le modèle, `cart.rhtml`, dans le même répertoire. Il est un peu plus complexe que d'habitude, aussi nous allons détailler.

Listing 7-21 Le modèle utilisé pour construire le XHTML interne du panier

```
<% cart.each do |product, qty| %> ❶
  <div class="cartItemsLine">
<% qty.times do |i| %> ❷
    <img src="product<%= product %>.png" class="cartItem" ❸
    id="item_<%= product %>_<%= i %>" style="position: relative" />
    <script type="text/javascript">
      new Draggable('item_<%= product %>_<%= i %>', { revert: true });
    </script>
  <% end %> ❹
    <span class="title">
      <%= PRODUCT_LABELS[product] + " (#{qty})" %>
    </span>
  </div>
<% end %> ❺
<%= 'Votre panier est vide.' if cart.empty? %> ❻
```

❶ Boucle sur les produits du panier, avec leur ID et quantité.

❷ Boucle de 0 à quantité – 1.

❸ On crée autant d'images du produit que nécessaire. Notez la classe `cartItem` et l'ID unique, utilisé dans le script pour pouvoir déplacer l'image.

❹ Fin de la boucle sur quantité.

❺ Fin de la boucle sur les produits.

❻ N'oublions pas les paniers vides !

Ouf ! Voilà un exemple massif ! Pour pouvoir le tester, lancez le serveur avec `ruby serveur.rb`, et naviguez sur http://localhost:8042/ :

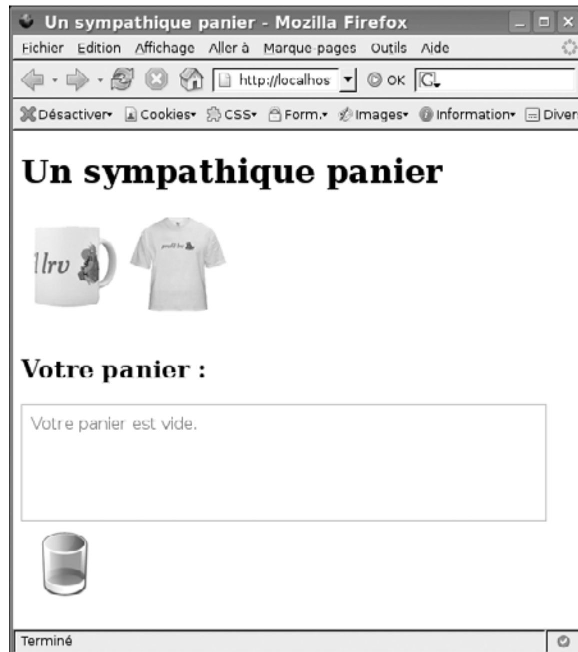


Figure 7–7
Notre panier, vierge, au démarrage



Figure 7–8
Déplacement d'un produit dans le panier pour commande



Figure 7–9
Le panier mis à jour



Figure 7–10
Le panier après quelques ajouts supplémentaires. Notez les quantités.



Figure 7–11
Retrait d'un produit du panier

Précautions d'emploi

Encore tous joyeux d'avoir réalisé un aussi joli exercice, vous imaginez déjà les interfaces riches que vous allez réaliser dans les tous prochains jours... Prêtez tout de même attention à ces quelques remarques importantes :

- Si vous deviez retirer une zone de dépôt du DOM, il est impératif que vous le signaliez au préalable au référentiel `Droppables`, à l'aide d'un appel à `Droppables.remove`. Si vous oubliez cette précaution, toute tentative de glisser-déplacer ultérieure échouera.
- Si vous avez une interface suffisamment (inutilement ?) complexe pour avoir des zones de dépôt imbriquées, vous devez absolument les enregistrer de l'intérieur vers l'extérieur, sous peine de ne voir réagir que les zones externes, aux dépends des zones imbriquées.

Tri de listes par glisser-déplacer

Inutile de nous arrêter là, le glisser-déplacer a une utilité de premier plan dans une interface web : l'ordonnancement d'éléments dans une liste ! Sans glisser-déplacer, on en est réduit à cliquer à qui mieux mieux sur des boutons de montée et de descente en face de chaque élément. Lorsqu'on n'a vraiment pas de chance, chaque clic

recharge la page. C'est comme ça qu'on obtient des listes non triées, la flemme ayant eu raison des bonnes intentions de l'utilisateur.

Mais avec le glisser-déplacer, tout va mieux, l'utilisateur retrouvant le confort habituel de ses applications classiques. C'est une utilisation tellement courante du glisser-déplacer que script.aculo.us fournit, évidemment, un mécanisme prêt à l'emploi : `Sortable`. Le mode d'emploi général est plutôt simple : on indique qu'il est possible de trier une structure avec un appel à `Sortable.create`, on désactive la fonction en appelant `Sortable.destroy`, et en cadeau bonus (parce que vous êtes sympathique), on obtient une représentation toute prête à l'envoi vers le serveur en appelant `Sortable.serialize` ! Voyons ces fonctions dans le détail.

Que peut-on trier ?

On peut théoriquement trier le contenu de n'importe quel élément de type bloc, à l'exception des conteneurs de table, comme d'habitude (plus spécifiquement les éléments `table`, `thead`, `tbody`, `tfoot` et `tr`). Il semble toutefois qu'en utilisant un `tbody` signifié comme conteneur, les éléments `tr` à l'intérieur puissent être ordonnés correctement sur MSIE 6/Win et Firefox.

Autre contrainte liée aux tables : un conteneur dans lequel on peut trier posera problème sur MSIE s'il est présent dans une table, à moins que la table ait une propriété CSS `position: relative`.

On peut donc trier les éléments fils d'une liste bien sûr (`ol`, `ul`) mais aussi de n'importe quel conteneur, par exemple des `p` dans un `div`. On peut même jouer sur l'horizontalité aussi, avec des éléments fils de type en ligne : `img`, `span`, etc. ou des éléments stylés en flottant, comme en témoigne d'ailleurs le spectaculaire puzzle d'exemple de http://wiki.script.aculo.us/scriptaculous/show/SortableFloatsDemo.

Activer les fonctions d'ordonnancement

Pour pouvoir manipuler le contenu d'un élément en vue d'un ordonnancement, il faut appeler `Sortable.create` sur cet élément conteneur. C'est sans doute l'appel script.aculo.us qui a le plus d'options jusqu'ici, à l'exception des effets noyau. Jugez plutôt !

Tableau 7–9 Options prises en charge par Sortable.create

Option	Description
`tag`	Nom des balises pour les éléments fils qui vont pouvoir être déplacés. Par défaut `li`, ce qui évite de le préciser dans le cas fréquent des listes. Pour les autres cas, à vous de le préciser.
`only`	Filtre supplémentaire optionnel pour les éléments fils qu'on peut déplacer, d'une syntaxe identique à l'option `accept` de `Droppables.add` : nom de classe CSS ou tableau de noms de classes.

Tableau 7–9 Options prises en charge par Sortable.create (suite)

Option	Description
`overlap`	Indique la direction de l'ordonnancement. Par défaut `'vertical'`, mais pour des éléments flottants ou des listes horizontales, préciser `'horizontal'`.
`constraint`	Identique à l'option homonyme de `Draggable`, mais ici à `'vertical'` par défaut. Pour retirer toute contrainte, on remettra donc à `false`.
`containment`	Détermine le ou les conteneurs au sein desquels les éléments fils peuvent être déplacés. Par défaut, concerne uniquement le contenu sur lequel on appelle `Sortable.create`. Mais on peut passer un tableau de conteneurs ou d'ID de conteneurs (qui doit obligatoirement inclure le conteneur courant), par exemple pour permettre l'échange entre plusieurs listes.
`handle`	Identique à l'option homonyme de `Draggable`.
`hoverclass`	Identique à l'option homonyme de `Droppables.add`.
`ghosting`	Identique à l'option homonyme de `Draggable`.
`dropOnEmpty`	Si actif, rend le conteneur `Droppable` lorsqu'il devient vide, en respectant `containment` pour les conteneurs sources. N'a pas d'intérêt si `containment` n'a que le conteneur courant, donc par défaut à `false`.
`scroll`	Analogue à l'option homonyme de `Draggable`, mais limité au conteneur courant, qui aura de préférence la propriété `overflow: scroll`.
`scrollSensitivity`	Identique à l'option homonyme de `Draggable`.
`scrollSpeed`	Identique à l'option homonyme de `Draggable`.
`onChange`	Fonction de rappel invoquée dès que l'ordre des éléments change. Reçoit l'élément déplacé en argument. Lors d'un transfert entre deux conteneurs, elle est appelée sur les deux.
`onUpdate`	Fonction de rappel invoquée en fin de glissement d'un élément, si l'ordre résultant a effectivement changé. Reçoit le conteneur en argument. Lors d'un transfert entre deux conteneurs, elle est appelée sur les deux. Les éléments fils doivent avoir des ID nommés comme l'exige `Sortable.serialize` (voir plus loin).
`delay`	Temps en millisecondes pendant lequel le bouton de la souris doit être enfoncé avant que le glisser-déplacer ne puisse avoir lieu. Vaut zéro (désactivé) par défaut.

Il existe par ailleurs deux nouvelles options depuis la version 1.6.1, `tree` et `treeTag`, qui permettent la gestion d'arborescences plutôt que de listes à plat. La documentation manque pour ces options, mais si le sujet vous consume d'intérêt, vous trouverez des exemples complets dans les tests fonctionnels fournis dans la bibliothèque.

Réalisons un premier exemple avec une seule liste. Copiez votre répertoire `draggable` dans un nouveau répertoire `sortable_one_list`. Nous allons modifier le HTML comme suit.

Listing 7-22 La page de test pour notre premier exemple d'ordonnancement

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Test de Sortable (une liste)</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,dragdrop"></script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test de <code>Sortable</code> (une liste)</h1>

<ul id="people">
    <li id="person_1">Adrien</li>
    <li id="person_2">Christophe</li>
    <li id="person_3">Élodie</li>
    <li id="person_4">Emmanuel</li>
    <li id="person_5">Marie-Hélène</li>
    <li id="person_6">Valérie</li>
</ul>

</body>
</html>
```

Un peu de CSS pour améliorer la présentation de la liste...

Listing 7-23 Quelques règles CSS font des merveilles sur notre liste

```
#people {
    padding: 0;
}

#people li {
    font-family: sans-serif;
    list-style-type: none;
    width: 20ex;
    height: 1.5em;
    line-height: 1.5em;
    padding: 0.5ex;
    margin: 0.5ex 0;
    background: #ffa;
    border: 1px solid #880;
    cursor: move;
}
```

Voyons à présent le script, outrageusement simple.

Listing 7-24 Notre script activant l'ordonnancement pour une liste

```
function initSortables() {
    Sortable.create('people');
} // initSortables

Event.observe(window, 'load', initSortables, false);
```

En dépit (ou plutôt à cause) de sa simplicité, ce code soulève une question ardue : est-il vraiment moral de faire payer le client pour ça ? Mais baste, laissons de côté ces considérations éthiques, et observons le résultat.



Figure 7–12
Déplacement dans une liste où le tri est possible

C'est sympathique, non ? Voyons un peu la différence avec le *ghosting* activé. Modifiez simplement les options de `create` :

```
Sortable.create('people', { ghosting: true });
```

Rechargez et tentez un déplacement.



Figure 7–13
Déplacement « ghost » dans une liste où le tri est possible

Vous voyez comme l'élément reste à sa place en attendant le relâchement, laissant un clone translucide servir au déplacement ? Je trouve ça plus agréable quand on transfère d'une liste à l'autre. Et justement...

Réalisons un deuxième exemple, avec deux listes aux éléments interchangeables ! Copiez votre répertoire dans un nouveau répertoire `sortable_two_lists`. On commence, comme d'habitude, par modifier le HTML.

Listing 7-25 Notre page HTML avec deux listes

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Test de Sortable (deux listes)</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,dragdrop"></script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>
```

```
<h1>Test de <code>Sortable</code> (deux listes)</h1>

<ul id="bulle" class="people">
    <li id="person_1">Adrien</li>
    <li id="person_2">Christophe</li>
    <li id="person_3">Emmanuel</li>
    <li id="person_4">Marie-Hélène</li>
    <li id="person_5">Valérie</li>
</ul>

<ul id="friends" class="people">
    <li id="person_11">Amir</li>
    <li id="person_12">Aurore</li>
    <li id="person_13">Claude</li>
    <li id="person_14">Élodie</li>
    <li id="person_15">Jacques</li>
    <li id="person_16">Juliette</li>
    <li id="person_17">Thomas</li>
</ul>

</body>
</html>
```

On va ajuster un tout petit peu la CSS pour traiter plusieurs listes similaires.

Listing 7-26 Notre CSS à jour pour traiter deux listes

```
.people {
    position: absolute;
    padding: 0;
}
.people li {
    font-family: sans-serif;
    list-style-type: none;
    width: 20ex;
    height: 1.5em;
    line-height: 1.5em;
    padding: 0.5ex;
    margin: 0.5ex 0;
    border: 1px solid #880;
    cursor: move;
}
#bulle li {
    background: #ffa;
}
#friends {
    left: 30ex;
}
```

```
#friends li {
    background: #cfa;
}
```

Et enfin, on ajuste notre script, ce qui permet de voir quelques options supplémentaires (au passage, on factorise, pour la première fois, les options).

Listing 7-27 Notre script pour deux listes aux éléments interchangeables

```
function initSortables() {
    var options = {
        dropOnEmpty: true, containment: [ 'bulle', 'friends' ],
        constraint: false, ghosting: true
    };
    Sortable.create('bulle', options);
    Sortable.create('friends', options);
} // initSortables

Event.observe(window, 'load', initSortables, false);
```

Voyons un peu le résultat...



Figure 7–14
Déplacement interliste, avec « ghosting » !

Remarquez que les éléments de liste prennent évidemment la couleur de leur liste conteneur, en vertu des règles CSS applicables.



Figure 7–15
Nos listes après quelques transferts...

Il nous reste toutefois un petit problème : nous avons beau avoir activé l'option `dropOnEmpty`, elle ne va pas nous servir à grand-chose. En effet, une fois une des listes totalement transférée dans l'autre, les dimensions calculées par le navigateur pour celle-ci se réduisent à zéro. Il n'y a donc plus rien sur quoi ramener des éléments !

Voici une proposition de solution : nous allons définir une classe supplémentaire applicable aux listes, qui leur donne une couleur de fond et de bordure identifiables, et nous allons définir des dimensions par défaut pour les listes, qui seront certes dépassées (notamment en hauteur) l'essentiel du temps (car la valeur par défaut pour la propriété `overflow` est `visible`), mais nous seront bien utiles une fois l'une des listes vide. Voici déjà les ajustements à la CSS, mis en exergue.

Listing 7-28 Ajustements à notre CSS pour gérer les listes vides

```
.people {
    position: absolute;
    padding: 0;
    width: 22ex;
    height: 2em;
}
.people.empty {
    background: #eee;
    border: 1px solid silver;
}
...
```

Mais cela ne suffit pas : certes, une fois la liste vide, elle occupera toujours un certain espace, et on pourra déposer dessus, mais faute de couleurs pour la rendre évidente, l'utilisateur ne saura pas qu'il peut y déposer quoi que ce soit. Or, la classe à appliquer doit l'être dynamiquement. C'est l'occasion rêvée de s'essayer à la fonction de rappel `onUpdate`.

Listing 7-29 Notre nouvelle fonction initSortables

```
function initSortables() {
    var options = {
        dropOnEmpty: true, containment: [ 'bulle', 'friends' ],
        constraint: false, ghosting: true,
        onUpdate: function(list) {
            if (0 == list.getElementsByTagName('li').length)
                Element.addClassName(list, 'empty');
            else
                Element.removeClassName(list, 'empty');
        }
    };
    Sortable.create('bulle', options);
    Sortable.create('friends', options);
} // initSortables
```

Tant qu'une liste n'est pas vide, rien ne change par rapport à tout à l'heure. Mais si une liste est vide, on voit alors sa position, ce qui nous suggère qu'on peut déposer dessus à nouveau.

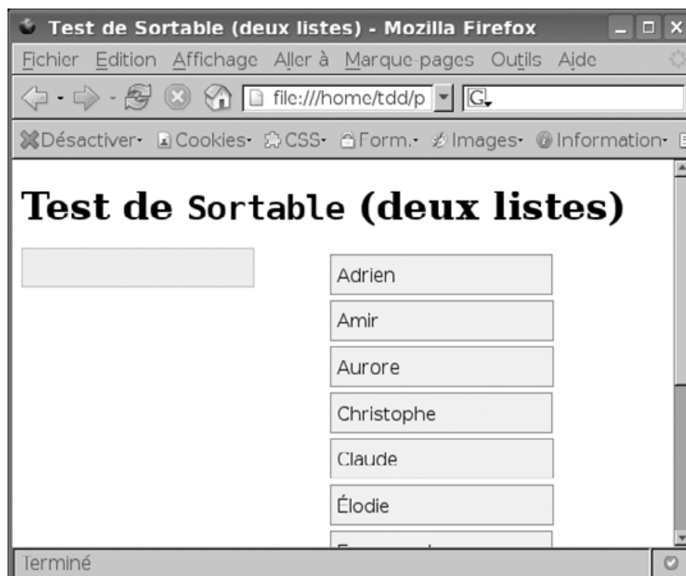


Figure 7–16
Une liste vide apparaît différemment.

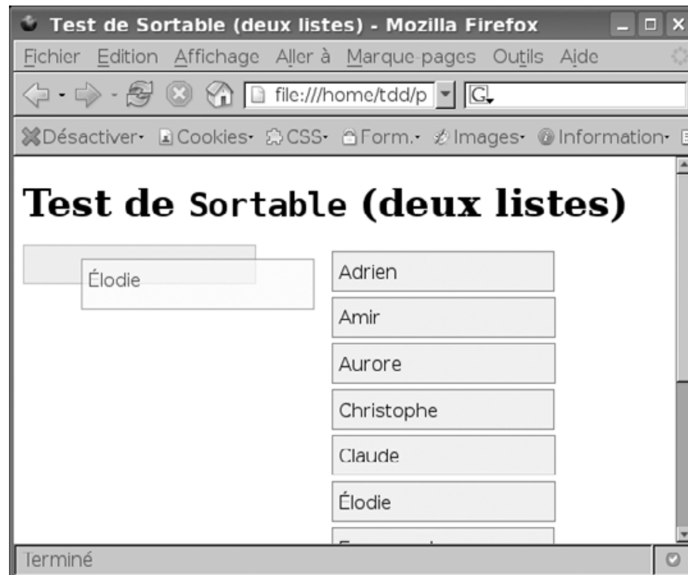


Figure 7–17
On peut donc facilement y redéposer un élément.

Ah, ça fait plaisir !

Désactiver l'ordonnancement

Dans la tradition désormais établie de présence d'un moyen de désactivation, symétrique à l'activation de fonctionnalité, il est possible de faire qu'un conteneur cesse d'offrir l'ordonnancement par glisser-déplacer. Il suffit d'appeler `Sortable.destroy` sur le conteneur qu'on avait passé à `Sortable.create`. Plus simple, je ne vois pas...

Envoyer l'ordre au serveur

Enfin, modifier l'ordre des éléments ne servirait pas à grand-chose si on ne pouvait pas envoyer l'information au serveur pour qu'elle persiste. Vous vous en doutez, script.aculo.us n'a aucune intention de vous laisser patauger dans l'examen du DOM pour construire manuellement la liste des ID correspondant au nouvel ordre.

Vous avez peut-être remarqué que dans nos exemples, nous avons attribué à chaque élément contenu (en l'occurrence des `li`) un attribut `id` de la forme `préfixe_suffixe`. C'est une exigence de `Sortable.serialize`, la fonction que nous pouvons utiliser pour obtenir déjà une représentation URL encodée de l'ordre de nos éléments. Cette fonction produit une chaîne constituée de paramètres `préfixe[]` dont les valeurs sont les suffixes concernés, dans le bon ordre.

Par exemple, pour le DOM correspondant au HTML suivant :

```
<ul id="bulle" class="people">
    <li id="person_1">Adrien</li>
    <li id="person_2">Christophe</li>
    <li id="person_3">Emmanuel</li>
    <li id="person_4">Marie-Hélène</li>
    <li id="person_5">Valérie</li>
</ul>
```

Un appel à `Sortable.serialize('bulle')` donnera ceci :

```
bulle[]=1&bulle[]=2&bulle[]=3&bulle[]=4&bulle[]=5
```

Cette syntaxe pour les noms résulte en un traitement automatique par la plupart des technologies côté serveur, PHP et Ruby On Rails pour ne citer qu'eux. En JSP, on est un peu plus malheureux, comme toujours pour la gestion des paramètres : on devra utiliser un `getParameterValues('bulle[]')` et itérer sur le résultat.

Notez que `serialize` accepte deux options.

Tableau 7–10 Options prises en charge par Sortable.serialize

Option	Description
`tag`	Identique à l'option homonyme de `Sortable.create`, et doit bien sûr être synchrone avec la valeur utilisée à l'appel de celle-ci.
`name`	Nom à utiliser pour les noms de champ de la représentation texte. Utilise par défaut l'ID du conteneur.

Complétion automatique de texte

Pour terminer avec les fonctions incontournables de script.aculo.us, penchons-nous sur ses possibilités de complétion automatique de texte. Il existe deux classes dédiées à ce domaine : `Ajax.Autocompleter` et `Autocompleter.Local`. Nous nous intéresserons ici à la première, la seconde permettant une complétion à partir de données stockées côté client, sous quelque forme que ce soit. Notez toutefois que les deux dérivent de `Autocompleter.Base`, et partagent bon nombre de paramètres. Dans les tableaux 7-11 des options et 7-12 des fonctions de rappel ci-après, j'indique en italique les paramètres spécifiques à la variante Ajax.

Création d'un champ de saisie à complétion automatique

Pour disposer d'un champ de saisie doté d'une complétion automatique de texte (champ généralement de type `<input type="text"... />`), il faut en réalité définir

deux éléments dans votre HTML : le champ lui-même, plus un conteneur destiné à recevoir les suggestions. Il vous appartient de styler comme bon vous semble ce conteneur, pour être conforme à la charte graphique de votre site. En effet, la couche serveur doit y retourner une liste non ordonnée (`ul/li`), afin que script.aculo.us puisse correctement gérer les interactions clavier et souris.

Voici un exemple de bloc HTML :

```
<input type="text" id="edtName" name="contactName" />
<div id="nameCompletions"></div>
```

Notez la concision : c'est en partie dû au fait que script.aculo.us ajoutera pour vous un attribut `autocomplete="off"` au champ de saisie (deux types de complétion automatique rentreraient en conflit), et ajoutera si nécessaire une propriété CSS `position: absolute` à votre conteneur de suggestions.

Côté JavaScript, vous avez simplement besoin de construire un objet `Ajax.Autocompleter` autour de ces deux éléments, accompagné éventuellement d'options :

```
new Ajax.Autocompleter('edtName', 'nameCompletions'[, options]);
```

Voilà, c'est tout simple. Nous allons mettre en œuvre dans quelques instants deux exemples, l'un trivial, l'autre plus personnalisé. Mais avant, faisons le tour des options et fonctions de rappel disponibles.

Voici la liste des options prises en charge par notre `Ajax.Autocompleter`.

Tableau 7–11 Options prises en charge par Ajax.Autocompleter

Option	Description
`paramName`	Nom du paramètre à envoyer au serveur. Par défaut l'attribut `name` du champ de saisie.
`tokens`	Permet la complétion incrémentale. Voir plus bas. Par défaut un tableau vide : `[]`, donc désactivé.
`frequency`	Intervalle d'examen pour complétion, en secondes. Par défaut `0.4`. Évitez de descendre en-dessous, pour des raisons d'ergonomie…
`minChars`	Nombre minimum de caractères à saisir avant de déclencher une complétion. `1` par défaut. 0 serait ignoré, mais mettre du négatif reviendrait à vous tirer dans le pied !
`indicator`	ID ou référence directe sur l'élément éventuel d'indication de progression (fortement conseillé), par exemple un `div` avec un *spinner* et un texte du style « assistance en cours… ». Rien par défaut.

Tableau 7–11 Options prises en charge par Ajax.Autocompleter (suite)

Option	Description
`select`	Nom de classe servant à filtrer le contenu de l'option retenue pour extraire la valeur de complétion. Pas de valeur par défaut : l'ensemble de l'option est prise, à l'exception des contenus marqués avec une classe CSS `informal` (voir plus bas, « Personnalisation des contenus renvoyés »).
`autoSelect`	Indique si lorsqu'un seul résultat est obtenu, il doit être automatiquement sélectionné. Vaut `false` par défaut.
`parameters`	Permet de définir des paramètres supplémentaires à envoyer au serveur pour complétion, afin d'ajouter généralement un contexte de recherche. Texte au format URL encodé. Vide par défaut.
`asynchronous`	Détermine si la requête est asynchrone (`true`, par défaut) ou non (`false`). Y toucher me semble une très mauvaise idée : une requête synchrone va geler le navigateur pendant son exécution...

Pour mémoire, seules les options en italique sont spécifiques à `Ajax.Autocompleter` : les autres viennent de `Autocompleter.Base`, et valent donc aussi pour `Autocompleter.Local`. Même chose pour les fonctions de rappel ci-dessous.

Il existe par ailleurs pas moins de cinq fonctions de rappel, qui sont toutefois très rarement utilisées, l'immense majorité des besoins étant déjà couverte par les options.

Tableau 7–12 Fonctions de rappel prises en charge par Ajax.Autocompleter

Fonction	Description
`callback`	Peut référencer une fonction chargée de modifier le texte du paramètre à envoyer au serveur, après composition initiale de celui-ci (`nom=valeur`). Il s'agit généralement d'ajouter un contexte fixe, ce qui se fait plus simplement avec `parameters`. Pas de valeur par défaut.
`onShow`	Référence la fonction appelée pour afficher la liste des suggestions. Par défaut, cale si besoin votre conteneur en position absolue immédiatement sous le champ de saisie, aligné en largeur, et le fait apparaître à l'aide d'un `Appear` de 1/8 de seconde.
`onHide`	Symétrique de `onShow`, qui par défaut masque le conteneur à l'aide d'un `Fade` de même durée.
`updateElement`	Appelée à la place de la fonction normalement chargée de placer dans le champ de saisie un texte correspondant à l'élément choisi dans la liste des suggestions.
`afterUpdateElement`	Appelée après l'insertion d'une valeur dans le champ de saisie suite à la validation d'une suggestion.

Dernier point avant d'aborder l'exemple, voyons comment script.aculo.us permet à l'utilisateur de faire son choix parmi les suggestions.

Interaction clavier et souris

À l'affichage du conteneur de suggestions, script.aculo.us sélectionne automatiquement la première, en lui ajoutant la classe CSS `selected` (à vous de styler comme bon vous semble). La bibliothèque gère alors, tout le temps de l'affichage, les manipulations suivantes :

- survol de la souris, touches curseur (Haut, Bas, Gauche, Droite) : ajustement la sélection ;
- clic, Entrée, Tabulation : validation de la sélection ;
- Échap : fermeture du bloc des suggestions, retour à la frappe normale ;
- les autres frappes clavier s'ajoutent normalement à la saisie, et mettent donc à jour la liste des suggestions.

On a donc l'embarras du choix, l'utilisation du bloc de suggestion est plutôt accessible.

Attention : j'ai constaté un problème de capture de la touche Entrée (préférer Tabulation) sur Opera 9 et Safari 2.

Un premier exemple

Pour notre premier exemple, nous allons créer un formulaire de saisie d'un nom de bibliothèque Ruby. De cette façon, nous avons d'emblée à notre disposition l'ensemble des modules de la bibliothèque standard, fournis avec Ruby. Qui plus est, cela nous donnera l'occasion d'écrire un code serveur un peu plus puissant que d'habitude.

Copiez un de vos répertoires de travail récents, par exemple `draggable`, dans le sous-répertoire `docroot` d'un nouveau répertoire `autocomplete_simple`. Nous n'aurons pas besoin de `dragdrop.js`, mais il vous faudra ajouter le `controls.js` fourni dans l'archive de script.aculo.us. Voici déjà le fichier HTML.

Listing 7-30 Le HTML pour notre complétion simple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Test simple de Ajax.Autocompleter</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,controls"></script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>
```

```
<h1>Test simple de <code>Ajax.Autocompleter</code></h1>

<form>
    <p>
        <label for="edtLibrary" accesskey="B">Bibliothèque Ruby</label>
        <input type="text" id="edtLibrary" name="library" />
        <div id="lib_suggestions" class="autocomplete"></div>
    </p>
</form>

</body>
</html>
```

Remarquez la simplicité. En temps normal, notre formulaire aurait bien sûr une action, un bouton `submit`, etc. Passons au script client, qui reflète admirablement la simplicité de notre exemple.

Listing 7-31 Le script client, vraiment trivial !

```
function initAutocompleter() {
    new Ajax.Autocompleter('edtLibrary', 'lib_suggestions',
        '/get_suggestions');
} // initAutocompleter

Event.observe(window, 'load', initAutocompleter);
```

Trois paramètres suffisent : l'ID du champ de saisie, celui du conteneur de suggestions (qui sera automatiquement masqué à ce moment-là), et l'URL à invoquer (sans aucun paramètre additionnel, c'est important). Pour le reste, nous laissons l'ensemble des options à leur comportement par défaut.

Ceci suffirait, mais on obtiendrait un aspect visuel pas très professionnel, qui ressemblerait à ceci.



Figure 7–18
La complétion sans style particulier

C'est justement pour fournir un style passe-partout plus agréable que nous avons ajouté une classe CSS à notre `div` conteneur (afin de pouvoir réutiliser le style sur plusieurs champs à complétion dans la même page). Voici notre feuille de styles, qu'on pourrait encore simplifier, mais qui fait dans un certain raffinement.

Listing 7-32 Une feuille de styles passe-partout de qualité pour les suggestions

```
div.autocomplete {
    position: absolute;     /* Histoire d'être explicite... */
    width: 250px;           /* Sera ajusté par script.aculo.us */
    background-color: white;
    border: 1px solid #888;
    margin: 0px;
    padding: 0px;
}

div.autocomplete ul {
    list-style-type: none;
    margin: 0px;
    padding: 0px;
}

div.autocomplete ul li {
    list-style-type: none;
    display: block;
    margin: 0;
    cursor: default;
    /* Et quelques finasseries... */
    padding: 0.1em 0.5ex;
    font-family: sans-serif;
    font-size: 90%;
    color: #444;
    height: 1.5em;
    line-height: 1.5em;
}

/* Rappel : script.aculo.us active une classe 'selected'
   lorsqu'un élément est sélectionné */
div.autocomplete ul li.selected {
    background-color: #ffb;
}
```

À présent, passons à la couche serveur. Nous avons besoin de deux fichiers : notre éternel `serveur.rb` et un modèle pour la génération de la liste non ordonnée à renvoyer à la couche client. Voici déjà le modèle.

Listing 7-33 Le modèle trivial de réponse, suggestions.rhtml

```
<ul>
<% libs.each do |lib| %>
    <li><%= lib %></li>
<% end %>
</ul>
```

Et maintenant le cœur de l'exemple, la couche serveur...

Listing 7-34 La couche serveur pour la complétion automatique

```
#! /usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

template_text = File.read('suggestions.rhtml')
suggestions = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/get_suggestions') do |request, response|
  name_start = CGI::parse(request.body)['library'][0]
  suffix = "/#{Regexp.escape(name_start)}*.rb" ❶
  libs = $LOAD_PATH.map { |dir| ❷
    Dir.glob(dir + suffix, File::FNM_CASEFOLD).map { |f|
      File.basename(f, '.rb')
    }
  }.flatten.sort.uniq
  response['Content-Type'] = 'text/html'
  response.body = suggestions.result(binding)
end

trap('INT') { server.shutdown }

server.start
```

La ligne ❶ nous protège de fragments comme « * » ou « .. », qui n'auront pas d'effet particulier. À la ligne ❷, `$LOAD_PATH` est une variable globale en Ruby qui contient l'ensemble des répertoires où chercher les modules. Pour chacun, on y cherche des fichiers démarrant par notre texte et portant l'extension `.rb`, sans se soucier de la casse. On « aplatit » le tableau de tableaux obtenu, on le trie, et on retire les éventuels doublons.

Tout ça en si peu de lignes !

Voyons à présent le résultat, en quelques étapes.



Figure 7–19
Le champ à vide, en attente d'un premier caractère saisi

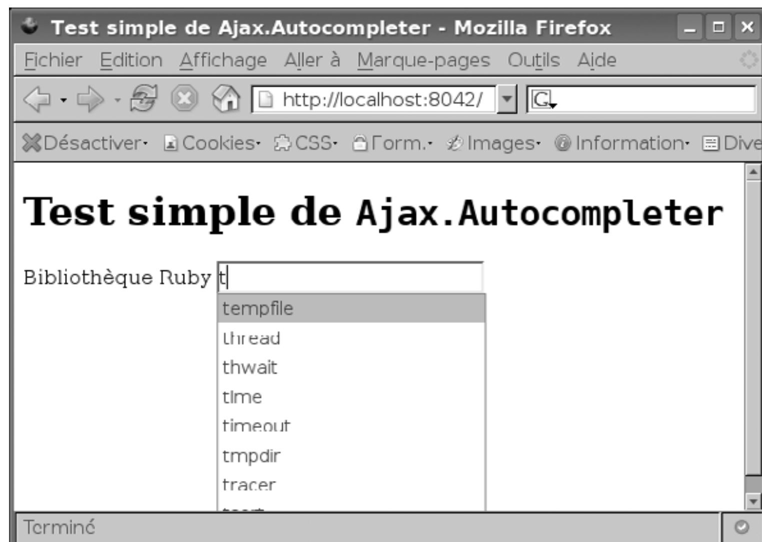


Figure 7–20
Complétion sur le premier caractère



Figure 7–21
Affinage avec un deuxième caractère saisi



Figure 7–22
La touche Bas (ou la souris) nous amène sur la seconde suggestion.

N'est-ce pas sympathique comme tout, franchement ? Bon, quand vous aurez fini de tester toutes les lettres de l'alphabet pour voir les modules de la bibliothèque Ruby standard (j'ai beau les connaître, je n'ai moi-même pas pu m'en empêcher...), passez à la prochaine section pour découvrir comment renvoyer des contenus plus riches comme suggestions.



Figure 7–23
Le clic, Entrée ou Tabulation valident la suggestion.

Personnalisation des contenus renvoyés

En décrivant l'option `select`, nous avons simplement brossé la surface de la sélection de contenu opérée par les `Autocompleters` lorsqu'on choisit une des suggestions.

Quelle que soit la configuration, seuls les contenus textuels (en termes DOM, les nœuds texte) sont retenus, agrégés les uns aux autres avec une espace entre chacun. Le corollaire est immédiat : vous pouvez renvoyer, dans chaque `li`, un contenu structuré (`div`, `p`, `img`, etc.) : seules les portions textes seront récupérées. Ainsi, considérez l'élément suivant :

```
<li>
    <img class="contactPhoto" src="/photos/contacts/17.jpg" />
    Thomas Fuchs
    <div class="note">Auteur de script.aculo.us</div>
</li>
```

Sa sélection aboutirait à l'insertion du texte « `Thomas Fuchs auteur de script.aculo.us` ».

Mais script.aculo.us ne s'arrête pas là. Il est fréquent de renvoyer des contenus textes plus riches que nécessaire, afin de fournir un contexte à l'utilisateur pour guider son choix. Par exemple, dans des suggestions de noms de contacts, on ajoutera l'adresse courriel ou le nom de la société (voire une photo, mais comme ce n'est pas du texte, elle serait de toutes façons ignorée à la validation).

Le moyen le plus simple pour gérer cela consiste généralement à ajouter aux segments « superflus » une classe CSS `informal`. Par défaut, script.aculo.us ignore les éléments ainsi marqués. Du coup, en modifiant l'exemple précédent comme ceci :

```
<li>
    <img class="contactPhoto" src="/photos/contacts/17.jpg" />
    Thomas Fuchs
    <div class="note informal">Auteur de script.aculo.us</div>
</li>
```

On obtiendrait simplement en validant notre choix : `Thomas Fuchs`. Cependant, ce n'est parfois pas la façon idéale de travailler, notamment si vous avez beaucoup d'éléments superflus dans votre suggestion. C'est pourquoi l'option `select` vous permet de préciser une classe spécifique qui marque, au lieu des éléments à ignorer, les éléments à inclure.

Dans l'exemple qui va suivre, nous allons tirer parti de cette fonctionnalité pour renvoyer des contenus complexes mais ne retenir que le nom de chaque suggestion pour l'ajout dans le champ de saisie. Nous allons aussi illustrer la complétion incrémentale, qui permet de saisir plusieurs valeurs à la suite, en les séparant par un *token*, qui est généralement la virgule. On ne dérogera pas ici à la règle.

Copiez votre répertoire `autocomplete_simple` dans un nouveau répertoire `autocomplete_advanced`. La page HTML nécessite juste quelques ajustements cosmétiques (titre, pluriel au nom du champ et à son libellé, etc.).

Listing 7-35 Notre page HTML mise à jour pour une complétion incrémentale

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Test avancé de Ajax.Autocompleter</title>
    <link rel="stylesheet" type="text/css" href="tests.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,controls"></script>
    <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test avancé de <code>Ajax.Autocompleter</code></h1>
```

```
<form>
    <p>
        <label for="edtLibraries" accesskey="B">Bibliothèques Ruby
        </label>
        <input type="text" id="edtLibraries" name="libraries" />
        <div id="lib_suggestions" class="autocomplete"></div>
    </p>
</form>

</body>
</html>
```

Le script est également mis à jour pour utiliser quelques options spéciales, afin notamment de continuer à envoyer un paramètre `library` alors que le champ, plus logiquement, s'appelle `libraries`.

Listing 7-36 Notre script ajusté

```
function initAutocompleter() {
    new Ajax.Autocompleter(
        'edtLibraries', 'lib_suggestions', '/get_suggestions',
        { paramName: 'library', tokens: [','], select: 'libName' });
} // initAutocompleter

Event.observe(window, 'load', initAutocompleter);
```

Pour chaque module Ruby, nous allons fournir sa taille et sa date de dernière modification. Nous aurons donc un modèle un peu plus riche.

Listing 7-37 Le modèle de suggestions avec les nouvelles données

```
<ul>
<% libs.each do |lib| %>
    <li>
        <div class="libName"><%= lib.name %></div>
        <div class="libMTime">
            ➥ <%= lib.mtime.strftime('%d/%m/%Y %H:%M:%S') %></div>
        <div class="libSize"><%= lib.size %> octets</div>
    </li>
<% end %>
</ul>
```

Évidemment, la CSS suit.

Listing 7-38 La feuille de styles ajustée pour ces nouvelles données

```
div.autocomplete {
    position: absolute;     /* Histoire d'être explicite... */
    width: 250px;           /* Sera ajusté par script.aculo.us */
    background-color: white;
    border: 1px solid #888;
    margin: 0px;
    padding: 0px;
}

div.autocomplete ul {
    list-style-type: none;
    margin: 0px;
    padding: 0px;
}

div.autocomplete ul li {
    list-style-type: none;
    display: block;
    margin: 0;
    cursor: default;
    /* Et quelques finasseries (mais plus de color)... */
    padding: 0.1em 0.5ex;
    font-family: sans-serif;
    font-size: 90%;
    height: 3.5em;
}

/* Rappel : script.aculo.us active une classe 'selected'
   lorsqu'un élément est sélectionné */
div.autocomplete ul li.selected {
    background-color: #ffb;
}

div.libMTime, div.libSize {
    font-size: 80%;
    color: #444;
}

div.libMTime {
    margin: 0.2ex 0 0 2ex;
}

div.libName {
    font-weight: bold;
}
```

```
div.libSize {
    margin: 0 0 0.5em 2ex;
}
```

Remarquez la classe spéciale pour la portion dont le contenu nous intéresse (libName) lorsque l'utilisateur fera son choix. Bien, il ne nous reste plus qu'à mettre à jour la couche serveur.

Listing 7-39 Notre nouvelle couche serveur, avec une classe dédiée

```
#! /usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

class LibInfo ❶
    attr_reader :name, :mtime, :size

    def initialize(name)
        @name = File.basename(name, '.rb')
        @mtime = File.mtime(name)
        @size = File.size(name)
    end

    def <=>(other) ❷
        self.name <=> other.name
    end
end

template_text = File.read('suggestions.rhtml')
suggestions = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/get_suggestions') do |request, response|
  name_start = CGI::parse(request.body)['library'][0]
  suffix = "/#{Regexp.escape(name_start)}*.rb"
  libs = $LOAD_PATH.map { |dir|
    Dir.glob(dir + suffix, File::FNM_CASEFOLD).map { |f|
      LibInfo.new(f) ❸
    }
  }.flatten.sort.uniq
  response['Content-Type'] = 'text/html'
  response.body = suggestions.result(binding)
end
```

```
trap('INT') { server.shutdown }

server.start
```

❶ Petite classe dont les attributs sont en lecture seule, représentant les données que nous retenons pour chaque module. Le constructeur prend le nom complet du module (chemin compris) en argument.

❷ L'opérateur <=>, en Ruby, est utilisé pour les comparaisons, donc pour le tri. Ainsi, on préserve le fonctionnement de l'appel ultérieur à `sort` et `uniq`.

❸ Et voilà ! Au lieu d'utiliser directement le nom simple du module, on construit un objet `LibInfo`.

Observons à présent un exemple de saisie en plusieurs étapes.



Figure 7–24
Saisie d'un premier caractère et suggestions riches



Figure 7–25
Choix d'une première valeur



Figure 7–26
La validation n'a retenu que le nom.



Figure 7–27
Après avoir saisi le « token » (la virgule), on attaque une nouvelle valeur.



Figure 7–28
La validation ajoute la suggestion retenue, au lieu de remplacer.

Et ce n'est pas tout ! Il y a d'autres services

Décidément, script.aculo.us nous facilite beaucoup de choses. Et pourtant, nous n'avons vu qu'une partie du framework. Citons rapidement les services restants :

- `Slider` permet de réaliser des champs alternatifs de saisie de valeur numérique basés sur une rampe (verticale ou horizontale) sur laquelle on peut faire évoluer des poignées. Totalement portable, le système n'en est pas moins hautement configurable : poignées multiples, limitations dynamiques des intervalles de valeurs, etc.
  - Essayez : http://wiki.script.aculo.us/scriptaculous/show/SliderDemo (et vous trouverez des exemples beaucoup plus poussés dans les tests fonctionnels inclus dans l'archive).
  - Documentation : http://wiki.script.aculo.us/scriptaculous/show/Slider
- *In Place Editing* fournit deux mécanismes permettant de modifier à la volée une portion quelconque de la page et de synchroniser côté serveur (comme les titres et notes dans Netvibes, par exemple). Les documentations en ligne sont très complètes (beaucoup d'options), sur http://wiki.script.aculo.us/scriptaculous/show/InPlaceEditing.
- `Builder` simplifie grandement la création dynamique d'éléments via le DOM. On a bien vu, au chapitre 3, que la création DOM d'un contenu complexe est vite fastidieuse. `Builder` simplifie considérablement cette tâche : http://wiki.script.aculo.us/scriptaculous/show/Builder.

Avoir le bon recul : les cas où Ajax est une mauvaise idée

Toujours dans l'optique de qualité et de méthodologie de ce livre, il est temps d'apporter un regard critique sur les trésors de possibilités nouvelles que nous offrent ces frameworks (sans parler des nombreux outils prêts à l'emploi qu'on peut trouver dans les frameworks supplémentaires, par exemple Dojo, évoqué plus loin dans ce chapitre).

Employer Ajax, des effets visuels ou des composants graphiques avancés n'est pas toujours une bonne idée : un usage à tout crin, sans considération ergonomique, peut rendre vos pages (et donc votre application web) inutilisable pour une partie significative des visiteurs et utilisateurs.

Nous allons d'abord évoquer l'ensemble des points à surveiller et des problèmes potentiels, pour finir par une série de pratiques recommandées qui permettent de les résoudre, ou au moins de les atténuer.

Ajax et l'accessibilité

On a déjà défini l'accessibilité dans la première partie de ce livre, en étudiant les relations entre celle-ci, JavaScript et le DOM. On l'a vu, l'accessibilité ne concerne pas que des utilisateurs non-voyants, comme on le croit souvent : elle concerne toute utilisation alternative de vos pages : périphériques à petit écran (de plus en plus répandus), sans clavier physique ou sans souris, handicaps visuels de toutes sortes (la cécité n'étant qu'un cas parmi d'autres), handicaps moteurs rendant l'utilisation du clavier ou de la souris difficiles, handicaps cognitifs comme la dyslexie...

Nous avons vu que JavaScript, et les pages dynamiques en général, ne vont pas forcément à l'encontre d'une accessibilité soignée. Mais il est évident que l'emploi de mécanismes encore plus dynamiques (effets visuels, glisser-déplacer, etc.) et le recours à Ajax soulèvent de nouvelles questions quant à l'accessibilité et l'ergonomie.

Considérations techniques

Nous avons d'abord la question de la disponibilité de `XMLHttpRequest`. Dans l'état actuel des navigateurs, disposer de `XMLHttpRequest` revient à avoir JavaScript activé et le DOM niveau 2 pris en charge, ce qui couvre la quasi-totalité des navigateurs répandus. Il existe tout de même un problème : le fait que sur MSIE 6 et antérieurs, `XMLHttpRequest` est fourni comme un ActiveX.

En effet, en raison de la très faible sécurité assurée par cette technologie, de très nombreux administrateurs système en entreprise désactivent purement et simplement ActiveX sur l'ensemble des MSIE de leur parc informatique. Sur ce type de poste client, même si nos pages peuvent utiliser JavaScript et manipuler le DOM, elles seront incapables d'utiliser Ajax. Ce n'est pas forcément un problème pour les applications intranet, ces mêmes responsables informatiques mettant généralement en place une politique plus tolérante pour la « zone intranet » reconnue par MSIE.

Le problème se corse encore davantage lorsque le service informatique recourt à une mesure encore plus draconienne, fréquente pour un parc où certains postes ne peuvent pas être verrouillés, et risquent donc d'avoir ActiveX activé : le filtrage au niveau du serveur mandataire (*proxy*). Ce filtrage passe généralement à la trappe tout fichier HTML ou JavaScript contenant le texte `new ActiveXObject`, ce qui a pour effet de supprimer le script entier ! Une portion significative, voire la totalité de votre logique applicative côté client est ainsi inutilisable.

Les lecteurs d'écran constituent également un point douloureux. Par essence, employer Ajax revient à mettre à jour une portion de la page plutôt que l'ensemble de la page. Or, si l'ensemble des études et tests menés ces deux dernières années montre quelque chose, c'est bien que les lecteurs d'écran sont encore très rudimentaires pour tout ce qui touche aux pages dynamiques.

Outre une pléthore de comportements dénués d'un quelconque effort de standardisation, pas un lecteur d'écran aujourd'hui disponible ne réagit correctement à un changement soudain d'une portion de la page, quand bien même cette portion serait à ce moment en cours de lecture, et en dépit de nombreux efforts dans le script visant à aider le logiciel à s'y retrouver (on peut par exemple consulter les tests édifiants menés en mai dernier par James Edwards et d'autres sommités de l'accessibilité sur les sept principaux lecteurs du marché, sur http://www.sitepoint.com/article/ajax-screenreaders-work).

Considérations ergonomiques

Et quand bien même tout va bien techniquement, la plus grande source de problèmes est probablement au niveau de l'ergonomie des sites. D'ailleurs, faut-il parler de sites ou d'applications ? La nuance peut sembler futile, mais elle joue un grand rôle dans l'esprit de l'utilisateur et dans ses attentes quant à l'ergonomie que vous proposez.

La plupart des internautes ont une association encore bien ancrée dans leur esprit :

navigateur = pages = site = interaction faiblarde

Par opposition, on a aussi :

application = écrans = programme (local) = interaction riche

Le terme application web (ou sa contraction *webapp*) ne suffisait pas à mélanger ces perceptions jusqu'à récemment, puisque les applications en question conservaient le schéma classique requête/chargement/réponse. Ajax et les frameworks JavaScript récents ont donné lieu à une nouvelle génération d'applications web, pour lesquelles une abréviation existe déjà : RIA, *Rich Internet Applications*, terme sur lequel se fixent de plus en plus les efforts marketing du Web 2.0. Il est vrai que Flash permettait de réaliser des interfaces riches depuis longtemps, mais de façon propriétaire et, jusqu'à récemment, plutôt restreinte en termes de communications avec le serveur (Flex est en train de changer cela, ceci dit). Flash est aussi souvent plus lourd et moins accessible que l'alternative Ajax.

Quoi qu'il en soit, tant que vos utilisateurs ne penseront pas à vos pages comme à une véritable application (et cela prendra du temps), ils n'auront pas les mêmes attentes. Là où il leur semble naturel, par exemple, de faire glisser une icône de document sur celle de la poubelle de leur bureau, ce même comportement ne leur vient pour l'instant pas à l'esprit lorsqu'ils sont dans un navigateur.

Ce hiatus peut poser plusieurs problèmes :

- Lorsque vous déclenchez une requête Ajax en arrière-plan, l'utilisateur ne perçoit pas forcément qu'il y a une requête en cours... Du coup, il peut être tenté d'utiliser l'interface d'une façon inappropriée (quitter la page ou la recharger, effectuer une action ailleurs sur la page qui risque d'entrer en conflit avec le traitement du résultat de la première requête, etc.).

- Dans la même série, l'utilisateur ne s'attend pas à pouvoir utiliser des mécanismes comme le glisser-déplacer dans une page web, à moins qu'on rende cette possibilité évidente d'une façon ou d'une autre.
- Les changements très localisés à la page ne sont pas forcément remarqués. L'utilisateur peut regarder son clavier en tapant ou avoir l'habitude plus ou moins consciente d'utiliser le rechargement de la page comme signe que le traitement a eu lieu : faute de rechargement visible, il ou elle n'a pas forcément le réflexe d'examiner à nouveau le contenu.

Utilisations pertinentes et non pertinentes

Indépendamment des attentes de l'utilisateur, il existe des contextes dans lesquels utiliser Ajax est tout simplement contre-intuitif, voire contre-productif, ou avoir des effets de bord indésirables.

Ainsi, on a deux types de pages dans une application web : les pages transitoires, dont l'état n'a pas vocation à persister, et les autres. Avant Ajax, chaque étape d'un processus, qu'elle représente un état transitoire ou un état stable (qu'on pourrait vouloir ajouter à ses marque-page ou dont on souhaiterait envoyer l'URL à un ami), était obtenu à l'aide d'un chargement de page.

Le bouton Précédent (ou la touche associée), bien connu des internautes, et très utilisé notamment par les utilisateurs non avancés, fonctionnait alors conformément à l'intuition : il nous ramenait à l'étape précédente (peu importe que l'application s'en offusque ou pas, ce n'est pas le sujet).

Mais si cette série d'étapes passe à Ajax, il n'y aura pas de rechargement de la page d'une étape à l'autre : l'entrée précédente dans l'historique de navigation reste la page visitée avant de démarrer le processus en question. Cela peut causer une confusion chez l'utilisateur, qui va naturellement cliquer Précédent s'il souhaite revenir à l'étape précédente, pour se retrouver à la place plusieurs pages en amont dans sa navigation !

Il convient donc de bien examiner si les différentes étapes constituent un état fondamentalement transitoire (par exemple des résultats de recherche, la saisie d'un commentaire dans un blog, une validation de données), auquel cas Ajax est parfaitement acceptable, ou s'il ne serait pas préférable de recourir à une navigation plus traditionnelle.

Cette distinction intervient aussi au niveau des URL des pages : s'il perçoit la vue courante comme quelque chose de stable, qui pourra être consultée ultérieurement, l'utilisateur peut vouloir en utiliser l'URL (pour un marque-page, pour l'envoyer à un ami, etc.). Dans un système Ajax, l'URL ne change évidemment pas tout au long des étapes, et l'utiliser plus tard nous amènera au début du processus, ce qui sera là aussi source de confusion.

Enfin, gardez à l'esprit que pour la majorité des formulaires (inscription, modification de profil, etc.) leur envoi manuel, à l'aide d'un traditionnel champ de type `submit` ou `image`, est parfaitement justifié : faire passer ces formulaires en envoi Ajax à la volée empêcherait l'utilisateur de l'explorer, de se faire une idée de sa saisie, ou d'interrompre sa saisie pour la compléter ou la reprendre ultérieurement. Dans la mesure où, la plupart du temps, vous ne fournissez pas de mécanisme d'annulation par la suite, un envoi forcé à la saisie serait perçu comme une contrainte, une gêne.

Pratiques recommandées

Voici l'essentiel des recommandations pertinentes qu'on trouve aujourd'hui au sujet de l'emploi d'Ajax dans les applications web. Pour le reste, deux règles d'or : faire preuve de bon sens et se mettre réellement à la place de l'utilisateur novice, qui découvre vos pages sans aucun a priori sur leur utilisation.

Principes généraux

Une règle de base, qui vaut pour toutes les pages, est de commencer par un fonctionnement traditionnel, sans JavaScript, reposant uniquement sur les éléments actifs habituels : liens et formulaires. Il ne faut pas pour autant perdre de vue qu'on va ensuite enrichir ce fonctionnement avec JavaScript et de l'Ajax, afin d'éviter notamment des erreurs de modularisation côté serveur. Mais en partant d'une conception 100 % passive côté client, pour ensuite l'enrichir avec de l'*Unobstrusive JavaScript*, on assure automatiquement une dégradation élégante.

Cette approche d'amélioration progressive (le terme anglais *progressive enhancement* a fait école) a trouvé un nom bien à elle, lorsqu'elle s'applique à l'ajout d'un fonctionnement Ajax : Jeremy Keith, figure de proue de JavaScript et du DOM, l'a baptisée *Hijax* (http://www.domscripting.com/blog/display/41). Je ne saurais trop vous recommander de procéder ainsi : le surcoût en temps est minime, et les bénéfices sont énormes.

Par exemple, l'application en ligne Backpack de 37 signaux (http://backpackit.com), en dépit d'une interface très riche et exploitant largement JavaScript, le DOM et Ajax, reste parfaitement utilisable sur des navigateurs textuels comme Lynx. GMail, en revanche, est d'une accessibilité très discutable (http://www.chaddickerson.com/blog/2005/10/18, puis choisissez l'unique article).

Autre point général important : sur un site public, pensez à bien spécifier dès la page d'accueil que vous utilisez JavaScript et Ajax, et faites un petit exposé sur les impacts ergonomiques. Cela aide grandement à ajuster convenablement les attentes de vos utilisateurs, et améliore leur ressenti sur votre application.

Ergonomie et attentes de l'utilisateur

- Pendant une requête Ajax, signifiez visuellement, de façon claire, qu'un traitement est en cours. Si certaines portions de l'interface pouvaient poser problème pendant ce temps-là, désactivez-les, comme nous l'avons fait dans notre exemple de réponse XML au chapitre 5.
- Si vous avez recours au glisser-déplacer, rendez la possibilité évidente, aussi claire que possible, et de préférence à la fois textuellement et graphiquement. Et bien entendu, il doit y avoir une façon alternative, accessible, d'obtenir le même résultat (de nombreux périphériques ne permettront pas de simuler le glisser-déplacer, et les personnes souffrant d'un handicap moteur rendant l'usage de la souris difficile auront le plus grand mal à utiliser le glisser-déplacer).
- Lorsque vous venez d'ajouter ou de modifier un élément de la page, mettez-le en exergue visuelle, par exemple avec un effet `Highlight` ou `Pulsate` (comme dans notre dernier exemple d'ajout de commentaires). Au cas où la partie mise à jour ne serait pas à l'écran (en raison d'un défilement, ou d'une bascule de l'utilisateur sur une autre application pendant le traitement), il existe aussi des techniques pour émettre un son. Par exemple, Campfire, une application de forums interactifs en ligne (http://campfirenow.com), émet un son discret mais reconnaissable lorsqu'un nouveau message arrive.
- N'utilisez jamais Ajax pour envoyer une saisie automatiquement, à moins que cela soit le comportement général approprié à votre application (par exemple, dans le cas de Netvibes), ou qu'il ne s'agisse que d'une sauvegarde temporaire permettant une reprise ultérieure (à la Web Forms 2.0) ou une résistante à la panne (par exemple, dans un passage de QCM en ligne pour un examen).
- Bien que je recommande plutôt de recourir à une navigation traditionnelle quand le besoin est apparent, il existe un moyen de gérer le bouton Précédent et de fournir des URL adaptées à l'état dans le cadre d'états transitoires de pages utilisant Ajax. Voyez l'article de Mike Stenhouse : http://www.contentwithstyle.co.uk/Articles/38
- Si vous souhaitez fournir à vos utilisateurs la possibilité d'accéder ultérieurement à un état de votre page qu'ils ont obtenu par manipulation Ajax, vous pouvez tout à fait créer votre propre solution, avec l'ensemble des paramètres nécessaires dans l'URL, comme le fait par exemple Google Maps avec son lien « Obtenir l'URL de cette page », en haut à droite de la carte. Il vous appartient alors de mettre cette fonction bien en évidence.

Cognitif/Lecteurs d'écran

- Tant pour les utilisateurs ayant des difficultés de concentration que pour ceux utilisant un lecteur d'écran, il est préférable de déclencher une requête Ajax manuellement, afin d'être prêt à examiner son résultat, que d'avoir des modifications

périodiques dont on ne décèle pas toujours l'exécution.
Si vous utilisez un mécanisme de type `PeriodicalUpdater`, il est alors intelligent de fournir une option, facilement accessible, permettant à l'utilisateur d'opter pour un déclenchement manuel (un bouton ou un lien couplé à un `Updater` classique, par exemple). Le choix pourrait être proposé en début de page à l'aide d'une case à cocher, par exemple.

- Pour les traitements Ajax consécutifs à la frappe (complétion automatique, etc.) il vaut mieux éviter de déclencher la requête immédiatement : il est préférable d'attendre un bref intervalle d'inactivité, afin de laisser l'utilisateur taper aussi loin qu'il le souhaite, à sa vitesse de frappe normale. En effet, en l'interrompant à chaque caractère ou presque avec une liste de suggestions, on ne réussit qu'à détourner sans cesse son attention et à augmenter, au final, son temps de saisie.
- Il peut être intéressant aussi de permettre à l'utilisateur d'être explicitement averti d'une mise à jour d'une portion de la page suite à un traitement Ajax, à l'aide d'un message d'avertissement (`alert`) par exemple. Cette technique à l'avantage d'amener automatiquement l'attention de l'utilisateur et le focus de l'éventuel lecteur d'écran sur la notification, tout en restaurant ensuite le focus de ce même lecteur d'écran là où l'utilisateur lisait auparavant.
  Pensez à rendre le message explicite, avec un contexte clair : préférez « Le nombre d'inscrits est désormais 430 » ou « Votre commentaire est désormais présent en fin de liste » à « Opération réussie »...
  Enfin, il doit bien sûr s'agir d'une option, car elle gênerait les utilisateurs n'en ayant pas besoin. Par ailleurs, il ne s'agit pas forcément d'une panacée : certains lecteurs d'écran ne liront pas automatiquement le texte du message !
- Dans le cas précis des lecteurs d'écran, dans la mesure où, on l'a vu, certains gèrent à peu près JavaScript, mais aucun ne réagit correctement à une modification partielle due à Ajax, la meilleure solution peut être d'offrir à l'utilisateur la possibilité de désactiver purement et simplement Ajax. Il ne s'agit pas de lui demander de manipuler les options du navigateur, mais bien de cocher par exemple une case, qui ajustera le fonctionnement de vos scripts et basculera donc sur une navigation traditionnelle. Si vous avez développé façon *Hijax*, comme vu plus haut, c'est facile à implémenter.

Pour conclure, voici quelques articles particulièrement intéressants sur l'implémentation d'une utilisation dégradable d'Ajax et l'utilisabilité des formulaires qui y ont recours :

- http://adactio.com/journal/959
- http://particletree.com/features/the-hows-and-whys-of-degradable-ajax/
- http://particletree.com/features/degradable-ajax-form-validation/
- http://www.baekdal.com/articles/usability/usable-XMLHttpRequest/

Autres frameworks reconnus

Nous avons concentré nos efforts autour des deux frameworks de premier plan dans l'univers Web 2.0 : Prototype et script.aculo.us. Mais il en existe de nombreux autres. Bien entendu, l'engouement actuel pour le phénomène Web 2.0 implique une quantité importante de sorties sans beaucoup d'avenir ou de qualité.

Néanmoins, on trouve certains frameworks assez répandus et de bonne qualité, dont l'approche est parfois très différente de celles que nous avons étudiées. Il vous appartient de vérifier l'étendue de leur compatibilité en termes de navigateurs : si la plupart prennent en charge MSIE, Safari et les navigateurs basés sur Gecko, tous ne fonctionnent pas totalement sur Opera et Konqueror, par exemple.

Voici, à mon sens, les trois principaux frameworks supplémentaires qui tirent leur épingle du jeu :

Dojo

Ce framework Open Source (double licence AFL et BSD) est un poids lourd (3,4 Mo avec les exemples) architecturé en paquets (au sens Java du terme), et ambitionne de couvrir les besoins avancés des applications DHTML. Il fournit donc une bibliothèque de *widgets* (composants graphiques, avec des interfaces toutes prêtes vers Yahoo! Maps, Google Maps, del.icio.us, etc.), et des modules pour travailler avec Ajax, le glisser-déplacer, les entrées/sorties, l'animation, la journalisation, etc. Il dispose d'une bonne documentation en ligne, de listes de diffusion et d'un wiki documentaire.

Version actuelle : 0.3.1. Site officiel : http://dojotoolkit.com

Mochikit

Ce framework fournit un ensemble de modules couvrant les fonctions classiques : Ajax, facilitation de JavaScript, itérateurs, manipulation avancée du DOM, glisser-déplacer, journalisation et effets visuels, mais aussi beaucoup de possibilités sur la gestion des couleurs (façon CSS3), un système de gestion événementielle de type signaux, et quelques utilitaires autour des dates et heures et du formatage numérique.

Version actuelle : 1.3.1. Site officiel : http://www.mochikit.com

OpenRico

Basé sur Prototype, ce framework fournit bien sûr des fonctions Ajax et de glisser-déplacer ainsi que certains effets visuels (déplacement, redimensionnement et fondu, mais aussi un mécanisme portable de coins arrondis, typique du Web 2.0), ainsi que

des « comportements » (*behaviors*), c'est-à-dire des composants graphiques de plus en plus répandus : un menu accordéon (http://openrico.org/rico/demos.page?demo=rico_accordion) et des tableaux à chargement dynamique, similaires aux `DataGrids` de ASP.NET.

Version actuelle : 1.1.2. Site officiel : http://openrico.org

Pour aller plus loin...

Site

Nous l'avons cité très souvent, le site officiel de script.aculo.us est contient de très nombreuses documentations et exemples : http://script.aculo.us

Groupe de discussion

Le groupe Google RubyOnRails-Spinoffs a été créé en août 2006 pour servir de centre d'aide technique, avec les avantages d'indexation et de recherche de Google. On y trouve des personnes de bon niveau, et Thomas Fuchs lui-même. http://groups.google.com/group/rubyonrails-spinoffs

Canal IRC

Enfin, il faut noter qu'on trouve souvent une réponse rapide et fiable sur le canal IRC dédié à script.aculo.us, hébergé sur l'incontournable serveur `irc.freenode.net`. Le canal se nomme tout simplement `#scriptaculous`.

TROISIÈME PARTIE

Interagir avec le reste du monde

Pour découvrir et maîtriser Ajax, nous avons eu recours à une couche serveur locale, facile à personnaliser pour nos tests. Dans le développement de vos applications, vous allez généralement utiliser un service qui vous est propre, hébergé sur vos serveurs : c'est une situation similaire.

Pourtant, la puissance du Web 2.0 tient aussi largement à l'interopérabilité des services et à la possibilité pour une page de recouper des informations d'origines diverses et d'exploiter des services fournis par des tiers.

Nous nous intéresserons à deux grandes catégories d'interactions.

Au chapitre 8, nous examinerons les fameux services web, tant ceux proposés au travers d'une interface SOAP (vous aurez alors besoin de votre couche serveur pour agir comme intermédiaire, ce protocole complexe n'étant pas pris en charge par des objets JavaScript), que ceux proposés au travers d'interfaces REST. Ces derniers sont de plus en plus fréquents, et peuvent être attaqués directement par vos pages, ce qui est un avantage significatif dans bien des cas !

Le chapitre 9 se penchera sur la syndication de contenus, principalement au travers des deux formats de flux dominants : RSS 2.0 et Atom 1.0. Il s'agit de documents XML ; vous pouvez donc les récupérer directement depuis votre page pour les manipuler et créer dynamiquement du contenu. C'est ce que font, par exemple, certains agrégateurs en ligne. On peut aisément imaginer un portail personnalisé agissant de même.

Grâce à ces perspectives, vous pouvez rendre vos pages plus riches en contenu, en possibilités et en intérêt. Et peut-être, si ce n'est déjà fait, réaliserez-vous que dans le Web 2.0, l'interopérabilité n'est pas un vain mot.

8

Services web et REST : nous ne sommes plus seuls

Les services web étaient censés permettre à n'importe quel logiciel de communiquer par Internet avec n'importe quel autre, quels que soient leurs langages de programmation respectifs.

La promesse n'est que péniblement tenue, au point qu'une sorte de retour aux sources s'est opéré, qui a découvert des trésors dans ce bon vieux protocole HTTP, et ne voue plus nécessairement un culte à XML. On fait maintenant des API REST et on tend même à les préférer à leurs équivalents SOAP.

Nous verrons dans ce chapitre à quoi correspondent ces technologies. Après avoir aperçu les horizons qu'elles nous ouvrent, nous réaliserons quelques exemples concrets, toujours au travers d'API REST, pour discuter avec Amazon.fr, The Weather Channel et Flickr.

Pourquoi la page ne parlerait-elle qu'à son propre site ?

Tout simplement pour des raisons techniques. Conceptuellement, il est parfaitement acceptable que votre page fédère du contenu qu'elle obtient dynamiquement depuis des services hébergés sur d'autres serveurs. C'est fondamentalement le rôle des services web.

Du temps des services web « classiques » (avec leurs deux tonnes nominales de SOAP pour dire bonjour), personne n'aurait osé imaginer que les communications allaient avoir lieu directement depuis la couche client. Cela aurait pourtant été formidable de décharger votre serveur. Cependant, SOAP est tellement complexe que c'était inimaginable, à moins de recourir à Flash, à une applet Java ou, sur MSIE, à un ActiveX (au prix d'une damnation éternelle).

Imaginez un peu les possibilités... Aujourd'hui, tout service digne de ce nom fournit des services web, et de plus en plus souvent une API REST (les services très à la pointe n'offrent d'ailleurs que REST). Regardez Netvibes : tout le contenu de la page et tous les services viennent de l'extérieur. Des pages de portail personnalisé, comme Google IG, sont entièrement basées sur ce concept.

Contraintes de sécurité sur le navigateur

Seulement voilà, les possibilités, vous n'êtes pas les seuls à les imaginer. Des personnes aux intentions moins honorables pourraient s'en servir pour contourner des mesures classiques de sécurité.

Imaginons par exemple que vous êtes sur votre poste en entreprise. Vous avez accès à l'intranet, lequel contient sans doute des informations confidentielles. Vous avez aussi accès à Internet, bien entendu. Et là, vous accédez à une page dotée d'un script qui tente de récupérer des informations sur `http://intranet/...` pour ensuite les transmettre à un site distant. Vous imaginez la suite !

C'est pour cette raison que sur certains navigateurs, JavaScript interdit l'accès à d'autres domaines que celui dont la page est issue. Il peut s'agir d'un script issu d'un domaine A tentant de manipuler les fonctions ou variables d'un autre script (fourni par un `frame` ou un `iframe`) issu d'un domaine B. Dans le cas qui nous occupe ici, il s'agit d'un objet `XMLHttpRequest` qui n'a pas le droit de communiquer avec un domaine tiers.

Suivant le navigateur, cette contrainte est plus ou moins prononcée. Il est souvent possible d'accéder à un *domaine parent*. Par exemple, un script exécuté par une page de `intranet.example.com` peut avoir accès aux ressources de `example.com`. C'est la fameuse exception à la *Same Origin Policy*, qui remonte à Netscape Navigator 2.0 (ce qui ne nous rajeunit pas). On utilise pour cela l'attribut DOM propriétaire

document.domain. Vous trouverez une explication plus détaillée de la S.O.P. à la page suivante : http://www.mozilla.org/projects/security/components/same-origin.html.

Certains navigateurs ne mettent en place aucune restriction, par exemple MSIE (qui n'est pas à une faille de sécurité près, il faut dire).

En revanche, les navigateurs Mozilla (Mozilla, Firefox et Camino) sont très stricts. Pour désactiver cette limitation, il faut fournir le script en tant que **script signé**. Il est aussi possible de configurer spécialement votre navigateur de développement pour éviter cela pendant les tests (mais cela ne remplace pas la signature pour le déploiement en production).

Vous trouverez tous les détails sur http://developer.mozilla.org/en/docs/Security, mais sincèrement, le jeu n'en vaut pas la chandelle : en production, cela ne marchera pas, en tout cas pas assez pour XMLHttpRequest. L'internaute qui utilise vos pages n'a probablement pas effectué ces réglages (il n'en a peut-être pas besoin s'il utilise un navigateur moins sécurisé). Le seul moyen de fournir une page utilisant ces droits étendus à un tiers inconnu consiste à fournir la page et ses scripts comme une **archive signée**.

Outre le fait que cela nécessite un certificat apte à signer du code (ce n'est pas toujours le cas des certificats utilisés pour HTTPS) et qu'il faut ensuite utiliser un outil spécifique (SignTool) à chaque mise à jour du contenu de l'archive, cela change radicalement la façon d'invoquer la page depuis le reste du site (l'URL a l'aspect suivant : jar:http://www.example.com/truc.jar!/page.html). Du coup, c'est incompatible avec les autres navigateurs...

Une « couche proxy » sur votre serveur

La seule solution véritablement portable, qui a de surcroît l'avantage de ne rien demander à l'utilisateur, consiste à mettre en place une couche intermédiaire sur votre serveur, qui se contente d'effectuer la requête qu'on lui donne, et de transmettre le résultat tel quel à votre script. C'est ce procédé que nous utiliserons au cours de ce chapitre. En effet, notre couche serveur n'est pas soumise aux contraintes de sécurité du navigateur : elle peut émettre des requêtes où bon lui semble (tant que les routeurs laissent passer, en tout cas).

Il existe tout de même une différence entre l'utilisation d'une simple fonction de proxy et le modèle traditionnel de traitement sur la couche serveur. Ici, le traitement est fait côté client : c'est plus léger pour le serveur, qui n'est qu'entremetteur, et cela ouvre des possibilités d'extensions pour vos pages, sur le même principe que les extensions Firefox.

Architecture de nos exemples

Nos trois exemples pour ce chapitre suivront tous le même principe, mais avec divers degrés de complexité et de raffinement. L'architecture fondamentale est la suivante :

- Notre page a besoin d'un document XML émis par un service tiers.
- Ne pouvant y accéder directement, elle passe par une fonction « proxy » proposée par notre couche serveur. C'est vers cette fonction que notre requête Ajax s'effectue.
- Le document obtenu est généralement trop complexe pour une extraction manuelle à l'aide des propriétés DOM ; on passera donc soit par une extraction XPath (comme nous l'avons déjà fait au chapitre 5), soit directement par une feuille de styles XSLT, pour obtenir un fragment XHTML représentant les informations qui nous intéressent.

Peut-être n'avez-vous jamais eu l'occasion d'utiliser XSL ou XSLT ? L'abréviation XSL signifie *eXtensible Stylesheet Language*. C'est la principale technologie de mise en forme d'un contenu XML, avec CSS. Cependant, là où CSS ne fait que réaliser une mise en forme à destination d'un média (imprimante, écran, vidéo-projection), XSL ouvre beaucoup plus de possibilités.

On considère traditionnellement que XSL est composé de deux technologies : XSL-FO (*Formatting Objects*, finalisée en 2001), une architecture permettant de réaliser des adaptateurs pour des formats spécifiques (le cas le plus courant étant PDF), et XSLT (*Transform*, finalisée en 1999), qui transforme le document source en un document quelconque, le plus souvent balisé lui aussi (XHTML, SVG, MathML...), bien que rien ne l'exige. Dans les deux cas, XSL utilise XPath pour extraire les données souhaitées du document XML source.

Afin de réaliser des extractions XPath et des transformations XSLT directement dans nos pages web, nous allons recourir encore une fois à Google AJAXSLT. En effet, la prise en charge de ces technologies par les navigateurs, et surtout leur accessibilité par JavaScript, sont très hétéroclites.

Vous trouverez davantage d'informations sur XSL, XSL-FO et XSLT aux adresses web citées en fin de chapitre. Les feuilles XSLT utilisées dans ce chapitre ne sont pas très compliquées.

Comme toujours, les exemples complets sont dans l'archive des codes source.

Comprendre les services web

Le terme « service web » est apparu il y a quelques années pour désigner un mécanisme « portable » de communication entre objets distants, une catégorie de logiciels appelés ORB (*Object Request Broker)*.

Les technologies ORB traditionnelles, CORBA, DCOM et RMI, se révélaient en effet insatisfaisantes pour la majorité des besoins. On leur reprochait en effet souvent les points suivants :

- Elles sont attachées à certains langages ou à des plates-formes particulières, en dépit d'efforts ultérieurs de portabilité.
- Elles utilisent un format propriétaire et binaire de transfert (débogage complexe).
- Elles utilisent le réseau d'une façon parfois trop complexe ou trop lourde à gérer (ports propriétaires nécessitant des configurations de routeurs, maintien de connexion et maintien d'état, etc.).
- Elles nécessitent la génération, souvent statique (à la compilation, pas à l'exécution), de classes intermédiaires (*skeletons* et *stubs*).
- Elles nécessitent un référentiel centralisé des objets disponibles, ainsi parfois qu'un référentiel des objets actifs.

En dépit des performances souvent élevées offertes par ces technologies, leur mise en œuvre était jugée trop complexe pour de nombreux cas, et surtout recelait toujours des difficultés de portabilité et de maintenance.

Les services web devaient initialement reposer sur un format et un protocole ouverts, stables, robustes et bien maîtrisés : XML et HTTP. On a aussi souhaité dès le départ éviter toute gestion d'état au niveau du protocole, afin de conserver les principaux avantages de HTTP : la capacité à monter facilement en charge, et la tolérance à la panne courte. Un format basé sur XML devait permettre la description d'un service : WSDL (*Web Service Description Language*). Le protocole dédié aux services devait donc être simple, portable, et facile à mettre en œuvre. On le baptisa SOAP, pour *Simple Object Access Protocol*.

Hélas, l'univers des standards WS (*Web Services*) a tellement enflé, et l'effet *Design By Committee* a tellement engendré de complexité, que cet acronyme est devenu une vaste farce. D'ailleurs, la version 1.2 de SOAP l'a officiellement retiré. SOAP est devenu SOAP, qui n'est plus un acronyme. Le protocole est en effet devenu atrocement complexe (avec l'ajout de possibilités d'utilisation de SMTP ou XMPP au lieu de HTTP, les pièces jointes, etc.), et entouré d'une pléthore d'autres formats et protocoles : UDDI pour les référentiels de services, et les WS-*, de plus en plus honnis, qui ne cessent de fleurir (WS-Addressing, WS-Security, WS-ReliableExchange, etc.). Le W3C compte six groupes de travail distincts sur le sujet !

Le résultat d'une telle surenchère est un schisme classique. D'un côté, on trouve les poids lourds de l'industrie, qui ont consacré des millions au développement de produits et services basés sur ces technologies, et les poussent en avant afin de récupérer leur investissement (sur ce point, les courriels circulant dans les listes de diffusion des membres du W3C sont édifiants). Dans le même camp, on trouve tous les projets à gros budget dont les directeurs techniques ont succombé au chant de sirène des poids lourds en question.

De l'autre côté, on a, comme d'habitude, le camp des partisans d'un développement agile, rapide, flexible, souple, et néanmoins performant. C'est le camp de ceux qui ont mis de côté XHTML 2.0 et XForms au profit des alternatives « XHTML 5 » (http://whatwg.org/specs/web-apps/current-work/) et Web Forms 2.0 (http://whatwg.org/specs/web-forms/current-work/) ; le camp de ceux qui préfèrent Ruby On Rails à l'univers J2EE... le camp, enfin, qui a proposé REST comme alternative au dogme « WS-* ».

Qu'est-ce qu'une API REST ?

REST est l'acronyme de *REpresentational State Transfer*. Ce terme en soi est particulièrement malheureux, car il ne renseigne en rien sur la nature concrète de l'approche. Fournir une API REST sert le même objectif que fournir une API SOAP : permettre à des codes externes de dialoguer avec notre service, de façon portable et robuste. Dans les deux cas, on se repose sur HTTP. Cependant, la similitude s'arrête ici.

Dans une API REST, l'application est exposée sous forme de **ressources** : un gestionnaire d'utilisateurs, un compte utilisateur, un tableau de bord, un mémo, une tâche à faire, une liste des fichiers joints, etc. Chaque ressource est accessible au travers d'une URL unique, utilisée pour tout : liens vers la page concernée, demande de modification, de suppression, etc.

Chaque ressource fournit une interface d'accès uniforme, constituée principalement de deux aspects :

- une série d'opérations correspondant à des verbes HTTP (principalement `POST`, `GET`, `PUT` et `DELETE`, qui correspondent aux opérations de création, de lecture, de mise à jour et de suppression, ce qu'on appelle classiquement CRUD : *Create, Read, Update, Delete*) ;
- une série de types de contenus MIME, en requête comme en réponse. Ceux en requête offrent juste plus de souplesse quant au format de transfert des paramètres, tandis que les types de réponse attendus peuvent changer radicalement la *nature* de la réponse fournie par la ressource.

REST constitue une approche client/serveur sans gestion d'état (en tout cas, pas par le protocole : comme en HTTP classique, l'utilisation de cookies ou d'un équivalent peut simuler l'état). Sa simplicité permet aussi l'utilisation pertinente d'un système de cache, ou l'enrobage dans des couches supplémentaires (par exemple une couche d'authentification).

L'avantage majeur de REST est sa simplicité, et la facilité avec laquelle une application web classique peut fournir une API REST rien qu'en prenant en charge quelques verbes HTTP supplémentaires dans sa couche serveur lorsque c'est pertinent.

Il est tout simplement impensable d'avoir un client SOAP en JavaScript : le code résultant serait bien trop lourd ; il faudrait recourir à un objet natif non standard, une applet Java ou, sur MSIE, un contrôle ActiveX.

En revanche, avoir un client REST est trivial avec `XMLHttpRequest`, au moins pour les verbes `GET` et `POST`. Les verbes suivants ne sont pas toujours pris en charge, mais le standard W3C en cours de finalisation les prévoit explicitement.

REST est donc en train de gagner de plus en plus de points dans l'opinion des développeurs (ce qu'on appelle en anglais la *developer mindshare*). La preuve : les API de gros services (Amazon, eBay, Yahoo!, Flickr, etc.) non seulement offrent du SOAP et du REST, mais mettent souvent l'accent sur REST, voire ne proposent *que* du REST ! Dans la même veine, le framework d'applications web Ruby on Rails, réputé pour son agilité, intègre REST au cœur de son architecture à partir de sa version 1.2, de façon presque transparente (« Pour toute application développée, une API offerte ! »).

Nous utiliserons donc, dans ce chapitre, exclusivement des API REST.

Cherchons des livres sur Amazon.fr

Pour notre premier cas concret, prenons une API particulièrement connue : celle d'Amazon. L'API est la même quel que soit le site utilisé (US, UK, France, Allemagne, etc.), mais certaines opérations ne sont pas disponibles partout. Nous utiliserons l'opération la plus courante : la recherche d'éléments. Pour être plus précis, nous allons nous spécialiser sur la recherche de livres.

À l'instar de nombreux fournisseurs de services, Amazon exige un enregistrement préalable pour pouvoir utiliser son API : cet enregistrement va vous fournir une **clé API**, que vous devrez indiquer à chaque requête au service.

Certaines API Amazon sont à utilisation payante, mais celle dont nous avons besoin ici est gratuite, dans la limite toutefois d'une requête par seconde (ce qui est largement suffisant pour les besoins de cet exemple !).

Commencez par vous faire un répertoire de travail, par exemple amazon.

Obtenir une clé pour utiliser l'API

Le portail des API Amazon présente l'ensemble des API disponibles, leurs mises à jour, etc. Il est sur http://www.amazon.com/gp/browse.html?node=3435361. On trouve également toutes les informations techniques, pour les développeurs, sur http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=5.

1. Sur la page portail, suivez le lien central Create (étape 1). Vous avez alors le choix entre associer votre compte Amazon Web Services avec un compte client existant, ou créer un compte dédié.
2. Supposons le second cas : saisissez une adresse courriel valide pour vous, et cochez l'option No, I am a new customer. Puis actionnez le bouton Continue.
3. Saisissez votre nom, confirmez l'adresse courriel et saisissez puis confirmez un mot de passe. Actionnez le bouton Continue.
4. Saisissez les informations demandées (oui, il y en a beaucoup, mais à notre connaissance, Amazon respecte bien la confidentialité...). Pour la section State, Province or Region, mettez par exemple F.
5. Prenez soigneusement connaissance des termes de la licence d'utilisation, en particulier la section 7A, qui concerne l'API *E-Commerce Service* (ECS), que nous utiliserons. Pour l'essentiel, en-dehors des dispositions classiques (s'abstenir d'utiliser des logos ou marques de partenaires Amazon, d'utiliser les données obtenues pour du marketing direct ou du spam, de modifier du contenu graphique fourni par Amazon, sauf pour redimensionnement...), il s'agit de se limiter à une requête par seconde et par IP, et de ne pas envoyer de fichiers supérieurs à 40 Ko...
6. Cochez la case indiquant votre acceptation de la licence et actionnez le bouton Continue.
7. Et voilà ! Vous allez recevoir un e-mail à l'adresse que vous avez fournie, en général en moins de deux minutes, intitulé *Welcome to Amazon Web Services*. Vous y trouverez un lien d'obtention de clé. Cliquez dessus.
8. Vous arrivez sur une page d'authentification. Saisissez l'adresse e-mail associée à votre compte et votre mot de passe. Actionnez le bouton Continue.
9. Vous y voilà ! Sur la droite de la deuxième section, vous voyez s'afficher vos deux identifiants personnels : la clé API (Your Access Key ID) et la clé secrète (Your Secret Access Key), qui ne s'affiche qu'après que vous avez cliqué sur le lien Show. Cette clé secrète est nécessaire pour « signer » certaines opérations de l'API. Nous n'en aurons pas besoin dans ce chapitre.
10. Copiez-collez soigneusement la clé API dans un endroit facile d'accès, comme un fichier texte dans votre répertoire de travail.

L'appel REST à Amazon.fr

Voyons à présent l'anatomie de notre interaction avec le service.

Anatomie de la requête

Notre requête utilise le verbe `GET` (celui des liens et des saisies manuelles dans la barre d'adresse), sur une ressource centrale pour Amazon Web Services, identifiée par l'URL suivante :

```
http://webservices.amazon.fr/onca/xml
```

Nous devons fournir deux paramètres fondamentaux :

1 L'API utilisée, avec le paramètre `Service`. Dans notre cas, on indiquera `AWSECommerceService`.

2 Notre clé API, avec le paramètre `AWSAccessKeyId`.

Par ailleurs, puisque nous travaillons avec ECS, nous devons fournir un paramètre supplémentaire :

3 L'opération qui nous intéresse, avec le paramètre `Operation`. Pour des recherches d'éléments à la vente, on spécifie `ItemSearch`.

Cette opération a de nombreux paramètres pour affiner les résultats, qui sont tous décrits dans la documentation de l'API, à savoir dans le cas qui nous occupe : http://docs.amazonwebservices.com/AWSEcommerceService/2006-06-28/ApiReference/ItemSearchOperation.html.

Sur les quelque 30 paramètres possibles (dont certains sont spécifiques à un type de produit, par exemple DVD), il y a seulement deux paramètres incontournables :

4 `SearchIndex`, qui précise le type de produit recherché. Dans notre cas, on utilisera `Books` ou `ForeignBooks`.

5 `Keywords`, qui contient la « saisie de l'utilisateur », donc les mots-clés à utiliser pour la recherche.

Un paramètre très important est `ResponseGroup`, qui précise les ensembles d'informations qu'on veut récupérer. Il existe 24 groupes possibles, certains étant une combinaison des autres. La valeur par défaut combine `Request` (qui nous renvoie le détail de notre requête, pour validation) et `Small` (la plus petite combinaison, qui ne fournit pas, par exemple, les prix ou les numéros ISBN, encore moins les images de couvertures).

Cela ne nous suffisant pas, nous définirons :

6 `ResponseGroup` à `Medium`.

Cette combinaison `Medium` ajoute aux valeurs par défaut (titre, auteurs et contributeurs, etc.) les groupes `ItemAttributes` (n° ISBN, nombre de pages, dimensions, informa-

tions de publication et de fabrication...), `OfferSummary` (tarifs proposés), `SalesRank` (position dans les ventes), `EditorialReview` (généralement constitué de la 4e de couverture et d'un ou deux extraits d'articles de presse) et `Images` (vignettes de couverture).

Voici donc un exemple d'URL (forcément très longue) pour une recherche « RSS Atom » en livres français. La valeur `APIKEY` est à remplacer par votre propre clé.

```
http://webservices.amazon.fr/onca/
xml?Service=AWSECommerceService&AWSAccessKeyId=APIKEY&Operation=ItemSearch
&SearchIndex=Books&Keywords=RSS%20Atom&ResponseGroup=Medium
```

Remarquez l'encodage URL de « RSS Atom », qui donne `RSS%20Atom`.

Allez-y, essayez-la dans votre navigateur : vous allez récupérer un document XML. Nous vous conseillons d'utiliser Firefox, qui l'affichera autrement mieux que MSIE, par exemple. En repliant certains nœuds du document XML résultat, on obtient l'affichage de la figure 8-1, qui permet de voir l'essentiel.

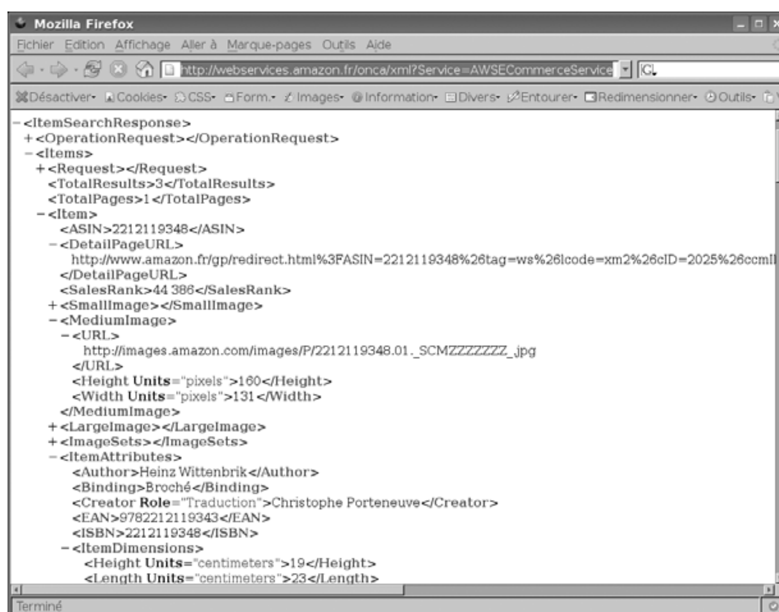


Figure 8–1
Une portion de notre document XML de résultats

Le document XML de réponse

Le nœud racine est `ItemSearchResponse`. Un nœud fils `Items` contient le nombre total de résultats et de pages, puis un nœud fils `Item` par résultat.

Chacun de ces nœuds `Item` propose entre autres :

- l'URL Amazon.fr du livre (`DetailPageURL`) ;
- la position dans les ventes (`SalesRank`) ;
- l'URL d'une vignette de taille moyenne (`URL` dans `MediumImage`) ;
- un élément de détails, `ItemAttributes`, avec entre autres :
- le titre (`Title`) ;
- l'auteur ou les auteurs (`Author`) ;
- les éventuels contributeurs, traducteurs... (`Creator`) ;
- le numéro ISBN (`ISBN`) ;
- le nombre de pages (`NumberOfPages`) ;
- la reliure (`Binding`) ;
- les dimensions (`ItemDimensions` et ses fils `Height`, `Length` et et `Width`) ;
- le prix éditeur (`ListPrice`) ;
- l'éditeur (`Publisher`) et la date de sortie (`PublicationDate`) ;
- le plus bas prix Amazon.fr (`LowestNewPrice` ou `LowestUsedPrice` dans `OfferSummary`).

Il nous appartient d'extraire ces informations et de les afficher agréablement. Ce sera le travail d'une feuille XSLT. Cependant, ne mettons pas la charrue avant les bœufs, et commençons par construire une page de recherche Amazon.fr !

Notre formulaire de recherche

Commençons par notre formulaire de recherche. Dans la meilleure tradition de l'amélioration progressive, garante d'accessibilité, nous allons réaliser un formulaire classique. Bon, idéalement, sa cible serait une action sur notre couche serveur qui récupérerait le contenu, le formaterait, etc. Ici pour simplifier, nous utiliserons directement l'URL de la ressource cible chez Amazon.fr.

Notre JavaScript interceptera l'envoi du formulaire pour utiliser plutôt Ajax, et réaliser une transformation côté client du XML récupéré.

Nous aurons ici une maigre couche serveur (juste un proxy de téléchargement), aussi utiliserons-nous notre structure habituelle : un répertoire `docroot` pour le contenu client, et le serveur au même niveau.

Dans le répertoire `docroot`, nous allons créer notre page HTML et sa feuille de style. Voici `index.html`.

Listing 8-1 Notre formulaire de recherche Amazon.fr

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Recherche Amazon (E-Commerce Service)</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Recherche Amazon (E-Commerce Service)</h1>

<form id="searchForm" method="get"
      ➥ action="http://webservices.amazon.fr/onca/xml">
    <p>
       <input type="hidden" name="Service" value="AWSECommerceService" /> ❶
        <input type="hidden" name="AWSAccessKeyId" value="…" />
        <input type="hidden" name="Operation" value="ItemSearch" />
        <input type="hidden" name="ResponseGroup" value="Medium" />
        <label for="cbxIndex" accesskey="C">Catégorie</label>
        <select id="cbxIndex" name="SearchIndex" size="1" tabindex="1"> ❷
            <option value="Books">Livres en français</option>
            <option value="ForeignBooks">Livres en VO</option>
        </select>
    </p>
    <p>
        <label for="edtKeywords" accesskey="M">Mots clés</label>
        <input type="text" id="edtKeywords" name="Keywords" tabindex="2" />
    </p>
    <p>
        <input type="submit" id="btnSearch" value="Chercher !" tabindex="3" />
    </p>
</form>

<div id="indicator" style="display: none;"></div>

<div id="results"></div> ❸

</body>
</html>
```

À partir de la ligne ❶, vous voyez la liste des paramètres que nous avons détaillés plus tôt. Évidemment, vous préciserez votre propre clé pour le paramètre `AWSAccessKeyID`. Il existe deux paramètres manipulables ici par l'utilisateur ❷ : le type d'élément recherché (livre en français ou livre en V.O.), et bien entendu les mots-clés de recherche. À l'issue de la recherche et du formatage du résultat, nous injecterons le fragment XHTML obtenu dans le conteneur d'ID `results` ❸.

Histoire que tout ceci soit un peu plus joli, nous allons y coller une feuille CSS.

Listing 8-2 Une petite feuille de style pour y voir plus clair

```
body {
    font-family: sans-serif;
}

h1 {
    font-size: 150%;
}

/* FORMULAIRE */

form#searchForm {
    border: 0.5em solid #888;
    background: #ddd;
    width: 60ex;
    padding: 0.5em;
}

form#searchForm p {
    position: relative;
    height: 2.2em;
    margin: 0;
}

#edtKeywords, #cbxIndex {
    font-family: serif;
    position: absolute;
    left: 14ex;
}

#edtKeywords {
    width: 45ex;
}

#indicator {
    margin-top: 1em;
    height: 16px;
    color: gray;
```

```
    font-size: 14px;
    line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
}
```

Ce formulaire (figure 8-2) doit déjà fonctionner tel quel : si vous saisissez une recherche et validez, vous devez obtenir un document XML Amazon.fr en réponse. Encore une fois, certains navigateurs, par exemple Firefox, affichent le XML de façon plus pratique que d'autres.



Figure 8–2
Notre formulaire de recherche

Passer par Ajax

Interceptons maintenant l'envoi normal du formulaire pour passer plutôt par Ajax. Pour ce faire, il va toutefois nous falloir une couche serveur jouant le rôle de proxy. Et en vertu de la *Same Origin Policy*, nous devrons accéder à notre page au travers de cette même couche serveur.

La couche serveur, intermédiaire de téléchargement

Cela nous donne un serveur plutôt simple, sans rien de nouveau par rapport aux précédents. Nous avons juste factorisé le code pour qu'il détecte automatiquement la présence d'un serveur proxy entre votre poste et Internet, à l'aide de la variable Linux/Unix habituelle `http_proxy`. Sous Windows, vous aurez besoin, avant de le lancer, de définir une variable adaptée. Par exemple, pour un proxy HTTP `proxy.maboite.com`, sur le port 3128.

```
set http_proxy=http://proxy.maboite.com:3128
```

Voici le code de `serveur.rb`, juste au-dessus de votre répertoire `docroot`.

Listing 8-3 Le code de notre serveur « proxy de téléchargement »

```
#! /usr/bin/env ruby

require 'net/http'
require 'uri'
require 'webrick'
include WEBrick

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

def get_requester
  if ENV.has_key?('http_proxy')
    comps = ENV['http_proxy'].split(':')
    host, port = comps[0..-2].join(':'), comps[-1]
    host.sub!(/^https?:\/\//i, '')
    port = port.to_i rescue 80
    Net::HTTP::Proxy(host, port)
  else
    Net::HTTP
  end
end # get_requester

server.mount_proc('/xmlProxy') do |request, response| ❶
  res = get_requester.get_response(URI.parse(request.query['url'])) ❷
  if res['Content-Type'] =~ /xml/ ❸
        response['Content-Type'] = res['Content-Type']
  else
        ct = 'text/xml'
        if res.body =~ /^<\?xml .*? encoding="(.*?)"\?>/
          ct += '; charset=' + $1
        end
        response['Content-Type'] = ct
  end
  response.body = res.body
end

trap('INT') { server.shutdown }

server.start
```

Notez l'URL locale pour notre fonction de proxy : `/xmlProxy` ❶, qui accepte un paramètre `url` ❷. Le bloc `if/else` en ❸ permet de forcer un contenu XML avec un jeu de caractères issu du prologue XML du document, au cas où le serveur d'origine ne renverrait pas le bon `Content-Type`.

Une fois ce serveur lancé, vous devez pouvoir accéder à votre page de recherche sur http://localhost:8042.

Intercepter le formulaire

C'est le moment de créer notre script client, déjà prévu dans notre HTML : `client.js`. Commencez par vérifier que vous avez bien copié `prototype.js`, depuis l'archive des codes source ou un de vos précédents répertoires de travail, dans `docroot`. Voici notre script.

Listing 8-4 Une première version de notre script, client.js

```
function bindForm() {
    Event.observe('searchForm', 'submit', hookToAJAX);
} // bindForm

function hookToAJAX(event) {
    Event.stop(event);
    var form = $('searchForm');
    $('indicator').update(''); ❶
    $('indicator').show();
    $('results').update(''); ❷
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(form.action) +
            '?' + encodeURIComponent(Form.serialize(form)), ❸
        onFailure: function() {
            $('indicator').hide();
        },
        onSuccess: function(requester) {
            $('indicator').update('Mise en forme&#8230;'); ❶
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var xml = requester.responseText; ❹
                $('results').update(xml.escapeHTML()); ❺
                $('indicator').hide();
            }, 10); ❻
        }
    });
} // hookToAJAX

Ajax.Responders.register({ onException: function(requester, e) {
    $('indicator').hide(); ❼
    alert(e);
}});

Event.observe(window, 'load', bindForm);
```

❶ L'indicateur a ici trois états : caché, visible avec juste l'icône (requête puis réponse), et visible avec l'icône et le libellé « Mise en forme... », qui indiquera tout à l'heure la transformation XSLT. C'est pourquoi, avant de le rendre visible, on vide son contenu HTML éventuel pour redémarrer sans libellé aux recherches suivantes.

❷ Dans le même esprit, au cas où ce n'est pas la première recherche, on vide le conteneur de résultats de son précédent contenu.

❸ Remarquez les précautions d'encodage pour s'assurer que toute l'URL ne constitue bien qu'un seul paramètre. Et c'est l'utilisation rêvée d'un `Form.serialize` (vous ne savez plus ce que cela fait ? courez au chapitre 4 !).

❹ Notre proxy de téléchargement garantissant un type MIME XML, le navigateur produira un DOM pour le XML renvoyé, et le mettra à disposition dans la propriété `responseXML` du requêteur. Comme on ne le transforme pas pour l'instant, prenons plutôt `responseText` pour pouvoir l'afficher.

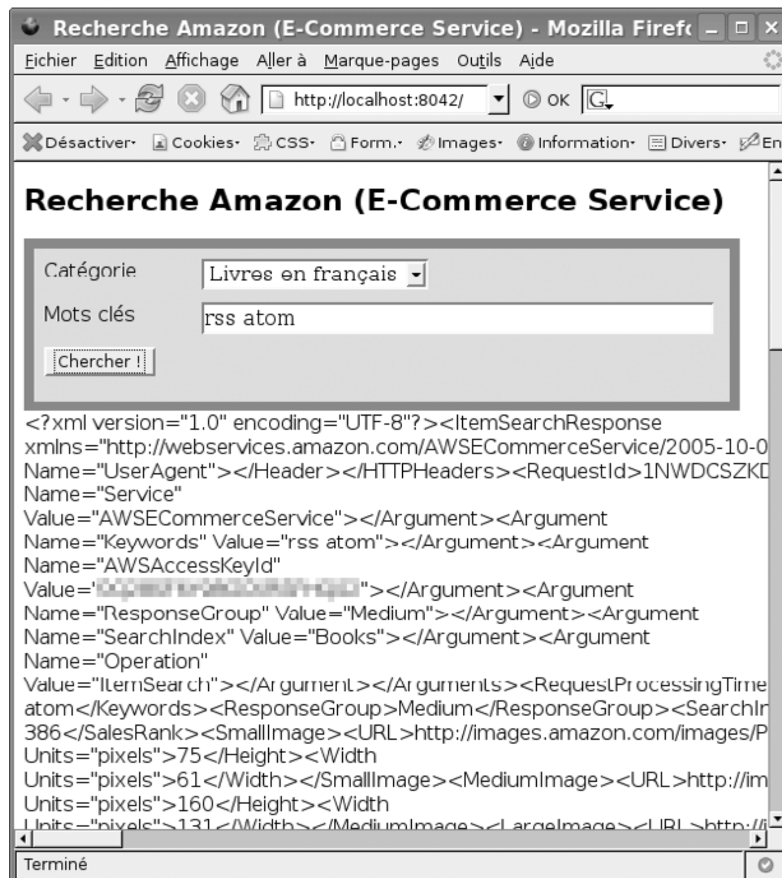


Figure 8–3
Le résultat injecté dans le conteneur results

❺ Nous ajusterons ce code à la prochaine section, en effectuant plutôt une transformation XSLT. Pour l'instant, nous visualisons directement le XML. Attention, il n'y a pas de retours chariot ; c'est une seule ligne, potentiellement très longue...

❻ Ce `setTimeout` de 10 millisecondes autour du code « lourd » (requête Ajax, traitement de la réponse) est une astuce courante pour s'assurer que le navigateur va bien prendre le temps (puisqu'il a 10 ms devant lui) d'afficher notre libellé tout frais pour l'indicateur.

❼ C'est la première fois que nous utilisons `Ajax.Responders`, qui a été décrit en fin de chapitre 6. Ainsi, nous affichons toute exception survenue lors d'un traitement Ajax. On ne sait jamais...

Allez, on rafraîchit la page en prenant soin d'ignorer le cache éventuel, et on retente une recherche. Vous devez normalement voir apparaître le résultat (figure 8-3).

Si vous examinez attentivement le XML retourné, vous apercevez effectivement les éléments que nous avons déjà vus : `Item`, `ItemAttributes`, `Title`, etc.

De XML à XHTML : la transformation XSLT

Utilisons maintenant une feuille de styles XSLT pour transformer, presque d'un coup de baguette magique, ce magma XML en un joli fragment XHTML. Pour cela, nous avons plusieurs étapes à suivre :

1. Écrire la feuille XSLT.
2. Doter notre page de capacités XPath et XSLT.
3. La télécharger côté client, au chargement de la page.
4. Changer le traitement du résultat XML : au lieu d'afficher le XML, on le transforme et on affiche le résultat final.
5. Augmenter la CSS et quelques astuces JavaScript pour embellir le XHTML obtenu.

Notre feuille XSLT

Créez un sous-répertoire `xsl` dans `docroot`, et mettez-y le fichier `books.xsl` suivant (ou reprenez-le de l'archive des codes source du livre, parce que c'est un beau pavé)

Listing 8-5 Notre feuille XSLT

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
        ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <xsl:apply-templates select="//Items"/> ❶
</xsl:template>
```

```
<xsl:template match="Items">
  <xsl:variable name="viewedItems" select="count(Item)"/>
  <xsl:variable name="totalItems" select="TotalResults"/>
  <xsl:choose>
    <xsl:when test="$viewedItems &gt; 0">
      <p class="resultCount">
        <xsl:value-of select="$totalItems"/> r&#233;sultat(s) ❷
        <xsl:if test="$viewedItems &lt; $totalItems">
          dont <xsl:value-of select="$viewedItems"/>
          list&#233;s ici.
        </xsl:if>
      </p>
      <dl>
        <xsl:apply-templates select="Item"/> ❶
      </dl>
    </xsl:when>
    <xsl:otherwise>
      <p class="resultCount">Aucun r&#233;sultat !</p>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template match="Item"> ❸
  <dt>
<xsl:element name="a"> ❹
<xsl:attribute name="href">
        <xsl:value-of select="DetailPageURL"/>
</xsl:attribute>
      <xsl:value-of select="ItemAttributes/Title"/>
</xsl:element>
  </dt>
  <dd>
    <xsl:element name="img">
      <xsl:attribute name="class">cover</xsl:attribute>
      <xsl:attribute name="src">
        <xsl:value-of select="MediumImage/URL"/>
      </xsl:attribute>
    </xsl:element>
    <ul>
      <li>
        Auteur(s) :
        <xsl:for-each select="ItemAttributes/Author
                  ➥ |ItemAttributes/Creator">
          <xsl:sort select="@Role"/>
          <xsl:element name="span">
            <xsl:attribute name="class">
              <xsl:choose>
                <xsl:when test="not(@Role)">author
                </xsl:when>
                <xsl:otherwise>contributor</xsl:otherwise>
              </xsl:choose>
```

```
            </xsl:attribute>
            <xsl:value-of select="."/>
            <xsl:if test="@Role"> (<xsl:value-of select="@Role"/>)
            </xsl:if>
            <xsl:if test="position() != last()">, </xsl:if>
          </xsl:element>
        </xsl:for-each>
      </li>
      <li>
        <xsl:value-of select="ItemAttributes/Binding"/>,
        <xsl:value-of select="ItemAttributes/NumberOfPages"/> pages,
        <xsl:value-of
           ➥ select="ItemAttributes/ItemDimensions/Height"/> &times;
        <xsl:value-of
           ➥ select="ItemAttributes/ItemDimensions/Length"/> &times;
        <xsl:value-of
           ➥ select="ItemAttributes/ItemDimensions/Width"/> (cm)
      </li>
      <li>
        Publi&#233; par <xsl:value-of
                              ➥ select="ItemAttributes/Publisher"/> le
       <span class="date"><xsl:value-of
                       ➥ select="ItemAttributes/PublicationDate"/></span>
      </li>
      <li>ISBN : <xsl:value-of select="ItemAttributes/ISBN"/></li>
      <li>
        Prix &#233;diteur : <span class="price"><xsl:value-of
                  ➥ select="ItemAttributes/ListPrice/Amount"/></span>.
        Prix Amazon.fr :
        <xsl:choose> ❺
          <xsl:when test="OfferSummary/LowestNewPrice">
            <span class="price"><xsl:value-of
                 ➥ select="OfferSummary/LowestNewPrice/Amount"/></span>.
          </xsl:when>
          <xsl:when test="OfferSummary/LowestUsedPrice">
            <span class="price"><xsl:value-of
                ➥ select="OfferSummary/LowestUsedPrice/Amount"/></span>.
          </xsl:when>
          <xsl:otherwise>N/D</xsl:otherwise>
        </xsl:choose>
      </li>
  <li>
    <xsl:variable name="availableCount" select="OfferSummary/TotalNew"/>
    <xsl:variable name="usedCount" select="OfferSummary/TotalUsed"/>
    <xsl:choose> ❻
      <xsl:when test="$availableCount &gt; 5">En stock</xsl:when>
      <xsl:when test="$availableCount &gt; 0">
      <xsl:value-of select="$availableCount"/> exemplaire(s) neuf(s)
    </xsl:when>
```

```
    <xsl:when test="$usedCount &gt; 0">
      <xsl:value-of select="$usedCount"/> exemplaire(s) d'occasion
    </xsl:when>
    <xsl:otherwise>Ce livre n'est pas en stock actuellement.
    </xsl:otherwise>
    </xsl:choose>
  </li>
      <li>
        Position dans les ventes :
        <xsl:value-of select="SalesRank"/>
      </li>
    </ul>
  </dd>
</xsl:template>
</xsl:stylesheet>
```

Eh bien, quel monstre ! Nous avons en effet souhaité faire un exemple un peu consistant... Si vous n'avez jamais fait de XSLT, vous aurez peut-être du mal à tout suivre. Nous vous recommandons vivement d'aller jeter un œil aux ressources en ligne citées en fin de chapitre.

Voici quelques précisions :

❶ Ces `xsl:apply-templates` appliquent des modèles (*templates*) à une série de nœuds XML extraits à l'aide de l'attribut `select`.

❷ L'entité numérique `#233` représente le « é », qu'il vaut mieux ne pas laisser littéral, car certains navigateurs n'utilisent pas correctement la déclaration de jeu de caractères dans le prologue XML de notre feuille XSLT, et croient qu'il s'agit d'UTF-8. L'entité numérique sera interprétée correctement.

❸ C'est ce `template` qui contient toute la description de transformation pour un élément de résultat (pour un livre, donc).

❹ On utilise `xsl:element` quand on veut générer dynamiquement les valeurs de certains attributs (par exemple `href` ou `src`) pour l'élément. Sinon, on peut utiliser l'élément directement, comme c'est le cas pour de nombreux `p`, `dl`, `dt`, `dd`...

❺ Un `xsl:choose` exprime une alternative, comme un `switch` en C/C++/Java/C# ou un `case…of` en Delphi. On utilise un ou plusieurs `xsl:when` (le premier qui correspond gagne, on ignore les suivants), et éventuellement un `xsl:otherwise` pour les cas restants. Ici, on gère les différents cas pour les prix : prix du neuf s'il existe, sinon prix de l'occasion s'il existe, sinon N/D (cas très rare).

❻ Même mécanisme, cette fois-ci pour le nombre d'exemplaires. Nous avons également décidé (arbitrairement, mais cela ressemble au comportement officiel du site) qu'à partir de 6 exemplaires neufs, on disait juste « en stock », tandis qu'en-dessous, on précise (cela incite à l'achat en cas d'hésitation).

Apprendre XSLT à notre page

Sous le répertoire `docroot`, créez un répertoire `ajaxslt` avec les fichiers que nous avions utilisés au chapitre 5 : `misc.js`, `dom.js`, `xpath.js`. Ajoutez-en un nouveau, que vous trouverez dans l'archive originale : `xslt.js`. Nous vous rappellons qu'il vous faut au minimum Google AJAXSLT 0.5 (vous avez des versions suffisantes dans l'archive des codes source).

Il nous reste bien sûr à charger ces scripts, en modifiant l'en-tête de notre page HTML comme ceci.

Listing 8-6 Le début modifié de notre index.html, qui charge AJAXSLT

```
<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="ajaxslt/misc.js"></script>
<script type="text/javascript" src="ajaxslt/dom.js"></script>
<script type="text/javascript" src="ajaxslt/xpath.js"></script>
<script type="text/javascript" src="ajaxslt/xslt.js"></script>
<script type="text/javascript" src="client.js"></script>
```

Et voilà, cela suffit, notre page parle XPath et XSLT ! Si seulement cela fonctionnait aussi simplement pour tous les lecteurs...

Charger la feuille XSLT au démarrage

C'est là que nous allons commencer à faire des choses rigolotes... D'abord, il faut charger la feuille XSLT au démarrage, et en stocker le DOM pour utilisation ultérieure (à chaque résultat de recherche). On utilisera donc... une variable globale (d'accord, ça, ce n'est pas rigolo).

En revanche, tant que ce DOM n'est pas disponible, lancer une recherche est périlleux : si celle-ci, par extraordinaire, aboutit avant notre chargement XSLT, la page sera incapable de réaliser la transformation. Nous allons donc désactiver initialement le bouton d'envoi du formulaire, et ne le réactiver qu'une fois la feuille XSLT chargée !

Commençons par modifier notre script. Ajoutons tout en haut notre déclaration de variable :

```
var xsltSheet;
```

La fonction `bindForm` est déjà appelée au démarrage de la page. Même si son nom n'est du coup plus très approprié (`initPage` lui irait mieux, mais `bindForm`, c'est déjà mieux que `gloubiBoulga`), nous allons lui ajouter une requête Ajax sur notre document XSLT. Une fois la feuille récupérée, le bouton d'envoi du formulaire est activé.

Listing 8-7 Ajout du chargement XSLT à notre initialisation

```
function bindForm() {
    new Ajax.Request('xsl/books.xsl', {
        method: 'get',
        onSuccess: function(requester) {
            xsltSheet = xmlParse(requester.responseText);
            $('btnSearch').disabled = false;
        }
    });
    Event.observe('searchForm', 'submit', hookToAJAX);
} // bindForm
```

Les lecteurs observateurs et futés (mais si, vous aussi !) poseront sans doute la question : mais pourquoi donc analyser manuellement `responseText` avec `xmlParse`, plutôt que d'utiliser directement `responseXML` ? Tout simplement parce que pour utiliser `responseXML`, il faut avoir la garantie que le serveur a renvoyé un type MIME XML pour notre fichier `books.xsl`. Ce n'est pas toujours le cas (cela dépend de sa configuration), et dans notre mini-serveur WEBrick, ce n'est *pas* le cas. Donc on se blinde, et on analyse la feuille manuellement.

Dernière touche : il faut désactiver par défaut le bouton d'envoi. Modifions donc sa définition dans `index.html`, comme ceci :

```
<input type="submit" id="btnSearch" value="Chercher !"
        ➥ tabindex="3" disabled="disabled" />
```

Effectuer la transformation

Allez, on y est presque ! Maintenant que nous avons une feuille XSLT disponible, et la capacité à l'utiliser pour une transformation, changeons notre traitement des résultats de recherche. Modifions donc la fonction `onSuccess` de notre requête Ajax, dans `client.js`.

Listing 8-8 Notre traitement de résultat XML, mis à jour

```
onSuccess: function(requester) {
    $('indicator').update('Mise en forme&#8230;');
    var tmr = window.setTimeout(function() {
        window.clearTimeout(tmr);
        var html = xsltProcess(requester.responseXML, xsltSheet);
        $('results').update(html);
        $('indicator').hide();
    }, 10);
}
```

Il nous reste un minuscule détail à régler : par défaut, Google AJAXSLT affiche un journal de ses principales étapes de traitement, dans un pavé en haut à droite de la page. Pour éviter cela, ajoutons en fin de script, juste avant les inscriptions d'observateurs, la ligne suivante :

```
logging__ = false;
```

Cette variable globale est définie par Google AJAXSLT, et vaut `true` par défaut. Passée à `false`, elle réduit les moteurs de traitement au silence.

Vous voyez ? Ce n'était pas si difficile (d'ailleurs, si la simplicité vous frustre, nous vous suggèrons de recoder `xsltProcess` ou d'oublier ce livre). Observons le résultat, après un rechargement strict de la page et une nouvelle requête (figure 8-4).



Figure 8–4
Nos résultats transformés !

Franchement, ce n'est pas si mal ! Mais on va améliorer tout ça...

Embellir le résultat

Commençons par ajouter quelques jolies choses à notre feuille CSS.

Listing 8-9 Ajouts CSS destinés aux résultats de recherche

```
/* RÉSULTATS DE LA RECHERCHE */

span.contributor {
    color: #666;
    font-size: 90%;
}

#results dt {
    font-weight: bold;
    font-size: 120%;
}

#results dd {
    margin: 0.5em 0 0.5em 1em;
    width: 50em;
    height: 160px;
    position: relative;
    padding: 0.25em;
    border: 1px solid gray;
}

#results dd ul {
    position: absolute;
    left: 150px;
    top: 5px;
}
```

Voyons ce que cela donne (figure 8-5)...

Il faut peu de chose... Ceci dit, peut-être avez-vous remarqué deux fausses notes dans ce joli résultat :

1. Les dates utilisent un format plutôt technique : 2006-06-08 par exemple. Ce n'est pas terrible...
2. Les prix sont prohibitifs (à moins qu'il ne s'agisse de roupies ou de livres turques) : 2 375 le livre RSS, on le sent passer ! En réalité, Amazon.fr, comme tout site de commerce électronique, stocke les prix en unités basses (ici les cents, ou centimes d'euros) pour éviter toute erreur d'arrondi.

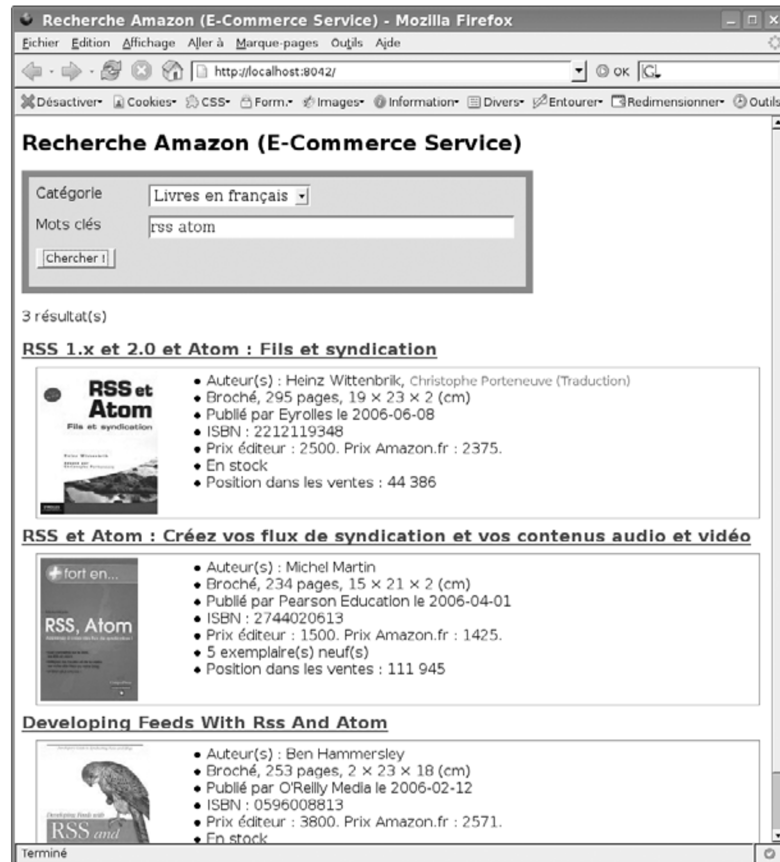


Figure 8–5
Avec un peu de CSS, tout va mieux…

Nous ne pouvons pas laisser la page comme cela. Il faut trouver une astuce. Nous pourrions entrer dans d'infinies « bidouilles » XSLT, qui fournit de nombreuses fonctions internes de découpe de texte. Seulement, d'une part cela ne permettrait tout de même pas de gérer la question de la date, et d'autre part, le moteur XPath de Google AJAXSLT ne tiendrait pas le coup : il n'implémente pas toutes ces fonctions avancées.

Non, le plus simple reste de recourir aux bonnes vieilles expressions rationnelles sur le XHTML obtenu. C'est d'ailleurs aussi pour cela que notre XSLT a pris soin d'isoler les prix dans des …, et la date de publication dans un ….

Créons donc une fonction adjustData, qui prend le XHTML original et renvoie le XHTML modifié. Les dates seront affichées joliment, du style « 4 avril 2005 » ou « 1er juin 2006 ». Les tarifs auront deux décimales et le symbole € en suffixe. Ajoutons tout cela au début de notre script client.js. Commençons par déclarer les

noms de mois, le format des prix et les expressions rationnelles d'extraction des textes à modifier.

Listing 8-10 Nos constantes de traitement des textes à localiser

```
MONTH_NAMES = [ 'janvier', 'février', 'mars', 'avril', 'mai',
    'juin', 'juillet', 'août', 'septembre', 'octobre',
    'novembre', 'décembre' ];
PRICER = new Template('#{euros},#{cents}&euro;');
RE_DATE = '(<span class="date">)([0-9-]+)(</span>)';
RE_PRICE = '(<span class="price">)([0-9]+)(</span>)';
```

Avant de voir le code de notre fonction `adjustData`, il est bon de rappeler quelques notions.

D'abord, nous allons utiliser la méthode étendue `gsub`, transmise aux `String` par Prototype. Cette méthode fournit une fonction pour produire le texte de remplacement à chaque correspondance de texte. Cette fonction récupère un objet représentant la correspondance, lequel objet est une sorte de tableau des groupes isolés par l'expression rationnelle. Par exemple, dans l'expression rationnelle suivante :

```
(<span class="date">)([0-9-]+)(</span>)
```

nous avons trois groupes, délimités par les parenthèses : le groupe 1, qui contiendra fatalement `<span class="date">`, le groupe 2, qui contiendra le texte de notre date, et le groupe 3, qui contiendra nécessairement `</span>`. Le texte que nous renvoyons doit préserver les groupes 1 et 3, afin de permettre, par exemple, une manipulation CSS ou JavaScript. On ne change que le groupe 2.

Pour convertir un texte `aaaa-mm-jj` dans le format que nous souhaitons, nous allons commencer par le transformer en un tableau de nombres `[a, m, j]`, que nous utiliserons pour obtenir le texte final.

Quant aux prix, il s'agit de retenir les deux chiffres de droite comme cents, et de garder ceux plus à gauche comme euros. Il suffit de synthétiser un objet avec deux propriétés `euros` et `cents`, contenant ces fragments textuels, pour pouvoir utiliser l'objet Prototype `Template` déclaré dans `PRICER`.

Ces points étant éclaircis, voici le code de notre fonction `adjustData`.

Listing 8-11 Notre fonction adjustData, qui retouche le XHTML produit

```
function adjustData(text) {
    text = text.gsub(RE_DATE, function(match) {
        var comps = match[2].split('-');
        comps = comps.map(function(s) { return parseInt(s, 10); });
```

```
        if (1 == comps[2])
            comps[2] = '1er';
        var date = comps[2] + ' ' + MONTH_NAMES[comps[1] - 1] +
                   ➥ ' ' + comps[0];
        return match[1] + date + match[3];
    });
    text = text.gsub(RE_PRICE, function(match) {
        var centsPos = match[2].length - 2;
        var price = {
            euros: match[2].substring(0, centsPos),
            cents: match[2].substring(centsPos)
        };
        return match[1] + PRICER.evaluate(price) + match[3];
    });
    return text;
} // adjustData
```

Déclarer la fonction ne suffit pas : encore faut-il l'utiliser. Insérons donc un appel dans notre fonction de rappel `onSuccess` pour la recherche.

Listing 8-12 Notre fonction onSuccess mise à jour

```
onSuccess: function(requester) {
    $('indicator').update('Mise en forme&#8230;');
    var tmr = window.setTimeout(function() {
        window.clearTimeout(tmr);
        var html = xsltProcess(requester.responseXML, xsltSheet);
        html = adjustData(html);
        $('results').update(html);
        $('indicator').hide();
    }, 10);
}
```

Un rafraîchissement strict et une nouvelle recherche plus tard, nous obtenons quelquechose de similaire à la figure 8-6.

Cette fois, ça y est ! Nous avons terminé. Alors, comment trouvez-vous cela ? C'est sympathique, non ? Précisons au passage que, si vous souhaitez faire effectuer le traitement XSLT par Amazon directement (par exemple, si vous avez une énorme XSLT et d'énormes grappes de résultats, et si vos postes clients sont des charrues préhistoriques), c'est possible, à l'aide du paramètre `Style`, qui précise une URL, accessible sur Internet, pour votre feuille XSLT. Pour ce chapitre toutefois, l'intérêt est de réaliser le traitement totalement sur le client (ce qui décharge votre couche serveur).

Bon, nous vous laissons jouer avec une bonne heure et le montrer avec exubérance aux collègues. C'est votre heure de gloire, vous l'avez bien méritée (surtout si vous avez tapé le XSLT à la main !).



Figure 8–6
Tout est dans les détails...

Quand vous aurez fini, rejoignez-nous à la prochaine section.

Rhumatismes 2.0 : prévisions météo

Nous allons reprendre le même mécanisme général, cette fois-ci sur un autre service très populaire : les prévisions météo. Nous utiliserons le service de The Weather Channel® (« TWC »), qui est plutôt fiable, détaillé, et a en outre l'avantage de proposer une couverture mondiale.

Théoriquement, l'utilisation de ce service est soumise à l'enregistrement d'une clé API, à fournir dans chaque requête. On s'enregistre à partir de la page suivante : https://registration.weather.com/ursa/xmloap/step1. Toutefois, en ajustant un tout petit peu l'URL utilisée, notamment en évitant de dire qu'on utilise le service dans le cadre de sa variante « clé », on obtient les mêmes données, plus vite, et sans clé !

Si ce type de comportement est pratique pour les tests, et peut vous éviter un enregistrement fastidieux, il n'est toutefois pas forcément licite, et nous ne garantissons aucunement qu'il fonctionnera indéfiniment.

Si vous devez utiliser ce service en production, prenez le temps de vous enregistrer, et respectez scrupuleusement les conditions d'utilisation qui, soit dit en passant, sont un peu contraignantes (par exemple, vous devez vous limiter à un lieu à la fois, indi-

quer que les données sont fournies par TWC, fournir un lien sur la page d'accueil de leur site, plus trois liens promotionnels ailleurs, être un service gratuit...).

Nous allons réaliser une page permettant d'une part de saisir une ville (le nom une fois complet, nous listerons les variantes possibles, par exemple « Paris, France » et « Paris, Illinois »), et d'autre part d'afficher les prévisions à quatre jours pour cette ville, une fois celle-ci validée. Par défaut, la page utilisera Paris, en France.

La figure 8-7 montre l'aspect que doit avoir la page.

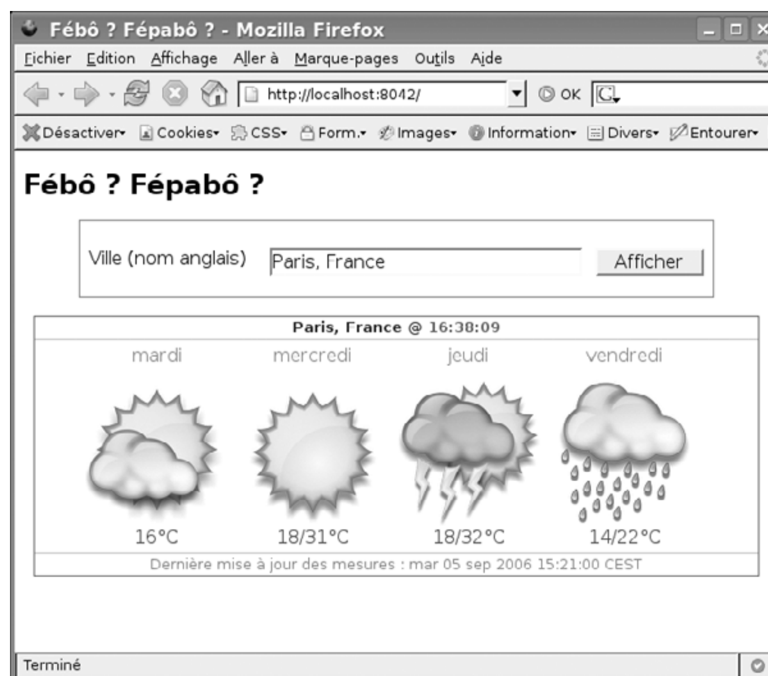


Figure 8–7
Notre page une fois terminée

C'est tout mignon, non ? Le jeu d'icônes de grande taille (160 × 160 pixels) utilisé, qui comprend près de 50 imagettes correspondant aux très nombreux codes (parfois redondants) de situation fournis par TWC, est quant à lui libre de droits : il s'agit d'un jeu d'icônes gratuit de Stardock. Il est disponible dans l'archive des codes source de ce livre. Vous trouverez des alternatives sur http://www.samurize.com/modules/ipboard/index.php?showtopic=3857. Elles sont toutes plus jolies que le jeu fourni par défaut dans le SDK de TWC...

Notez que l'heure affichée dans la partie haute est l'heure **locale** au lieu indiqué. Afin d'illustrer quelques petites finesses, nous ferons en sorte qu'elle soit mise à jour automatiquement par la suite (toutes les secondes), comme une horloge. La page recharge

automatiquement les prévisions toutes les 2 heures, ce qui correspond à l'intervalle de mise à jour des prévisions à 4 jours comme indiqué dans le SDK de TWC.

Préparer le projet

Commencez par créer un répertoire de travail `meteo`. Recréez-y les incontournables : notre `serveur.rb` (strictement le même que pour notre exemple Amazon.fr), le répertoire `docroot`, ses fichiers `prototype.js` et `spinner.gif`, et son sous-répertoire `ajaxslt` avec son contenu.

Nous utiliserons bien entendu des fichiers `client.js` et `client.css`, que nous réécrirons complètement en revanche. Vous pouvez partir du `index.html` de notre exemple Amazon.fr pour réaliser celui de cet exemple. Dans la même série, on pourra reprendre le squelette externe de `xsl/books.xsl` pour nos *deux* feuilles XSLT dans cet exemple : `search.xsl` et `forecast.xsl`.

Commençons par poser la page HTML et son style de base. Nous avons mis en axergue les changements par rapport à Amazon.fr.

Listing 8-13 Une première version de notre page index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Fébô ? Fépabô ?</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="ajaxslt/misc.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt/xpath.js"></script>
    <script type="text/javascript" src="ajaxslt/xslt.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Fébô ? Fépabô ?</h1>

<div id="results"></div>

<div id="indicator" style="display: none;"></div>

</body>
</html>
```

Notre CSS est pour l'instant très basique.

Listing 8-14 Une première CSS basique, client.css

```
* {
    font-size: 1em;
}

body {
    font-family: sans-serif;
}

h1 {
    font-size: 150%;
}

#indicator {
    position: absolute;
    left: 1em;
    bottom: 1em;
    height: 16px;
    color: gray;
    font-size: 14px;
    line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
}
```

Notez que, cette fois-ci, nous positionnons l'indicateur de progression en bas à gauche de la page. C'est un parti pris simple, qui nous permet de toujours avoir l'indicateur au même endroit alors que le reste de la page va varier en taille, ce qui évitera par exemple un décalage du bloc de prévision lorsqu'on recherche des correspondances de lieu...

Nous voici avec une page qui ne fait rien et le fait bien. Il est temps de lui donner un peu de nerf...

Récupérer les prévisions d'un lieu

Nous allons pour l'instant définir le code du lieu « en dur » dans le script. Il ne tiendra qu'à nous de le faire varier par la suite, à l'aide d'une sélection de lieu auprès de TWC.

Requête et réponse REST

L'API REST de TWC est très simple. Nous allons utiliser une première ressource, qui correspond au lieu voulu. Ce lieu est identifié par un code spécifique. Pour Paris en France, ce code est `FRXX0076`. L'URL de la ressource est donc http://xoap.weather.com/weather/local/FRXX0076. Nous y ajoutons un paramètre `dayf` (*days of forecast*) qui indique le nombre de jours de prévisions, et un paramètre `unit`, très utile, pour récupérer des valeurs en système métrique (°C, vitesses en m/s ou km/s, etc.). Au final, notre URL a l'aspect suivant :

```
http://xoap.weather.com/weather/local/FRXX0076?dayf=4&unit=m
```

La grappe XML retournée n'utilise pas un type XML, ce qui est une faute de configuration de TWC. Il nous faudra l'analyser manuellement avec `responseText` et `xmlParse`, comme pour nos feuilles XSLT. Elle a l'aspect suivant.

Listing 8-15 Vue partielle du résultat XML de demande de prévisions

```
<weather ver="2.0">
  …
  <loc id="FRXX0076">
    <dnam>Paris, France</dnam> (1)
    <tm>4:57 PM</tm> (2)
    …
  </loc>
  <dayf>
    <lsup>9/5/06 3:21 PM Local Time</lsup> (3)
    <day d="0" t="Tuesday" dt="Sep 5"> (4)
      <hi>N/A</hi> (5)
      <low>16</low>
      …
     <part p="d"> (6)
       <icon>44</icon> (7)
       <t>N/A</t> (8)
        …
      </part>
      <part p="n"> (6)
        <icon>33</icon>
        <t>Mostly Clear</t> (8)
        …
      </part>
    </day>
    <day d="1" t="Wednesday" dt="Sep 6">
      <hi>31</hi>
      <low>18</low>
      …
    </day>
```

```
    <day d="2" t="Thursday" dt="Sep 7">
    …
    <day d="3" t="Friday" dt="Sep 8">
    …
  </dayf>
</weather>
```

Ce n'est pas une grappe compliquée. Voici quelques précisions pour bien comprendre son fonctionnement :

❶ `weather/loc/dnam` donne le nom complet du lieu.

❷ `weather/loc/tm` donne l'heure locale du lieu. Afin de pouvoir manipuler cette heure par la suite (notre fameuse horloge), nous devrons interpréter correctement cette chaîne de caractères pour en faire un objet JavaScript `Date`.

❸ `weather/dayf/lsup` donne le moment de dernière mise à jour des données de mesure, en heure locale du lieu. Nous devrons là aussi extraire cette information pour en faire un objet `Date`, afin de pouvoir le reformater selon la langue active du navigateur, ce qui sera plus agréable pour l'utilisateur.

❹ Chaque jour fait l'objet d'un élément `day` dans `weather/dayf`. L'attribut `d` est un numéro d'ordre (0 est aujourd'hui, etc.). Bien que dans la pratique, cela semble être toujours dans l'ordre, rien ne le garantit dans la documentation : nous trierons donc dans la XSLT, à tout hasard. Notez aussi l'attribut `t`, qui contient le nom anglais du jour de la semaine. Ce serait pénible à obtenir autrement, alors on utilisera une astuce JavaScript pour transformer en noms français.

❺ Dans un `day`, les éléments `hi` et `low` donnent les températures maximale et minimale. L'unité est ici le °C, puisque nous avons demandé un résultat en système métrique. Notez que parfois, on n'a pas l'information complète (il arrive même que les *deux* soient `N/A`). Il faudra le gérer dans la XSLT.

❻ Chaque `day` a deux parties de détail, sous forme d'éléments `part`. Leur attribut `p` indique la période (`'d'` pour le jour, `'n'` pour la nuit). Par souci de simplicité, nous n'utiliserons que les parties `'d'`. Il est vrai que nous pourrions utiliser un paramètre XSLT pour indiquer si on souhaite, pour le premier jour (aujourd'hui), avoir l'information diurne ou nocturne, en fonction de l'heure locale du lieu… Nous vous laissons jouer avec cela !

❼ Dans une `part`, l'élément `icon` indique la représentation graphique à utiliser pour les conditions météo. La valeur va de 0 à 47. Vous trouverez dans l'archive des codes source, dans le répertoire de cet exemple, un répertoire `icons` avec tous les fichiers PNG correspondants, que vous pouvez reprendre et placer dans votre `docroot` si vous réalisez l'exemple vous-même.

❽ Enfin, l'élément `t` d'une `part` fournit un libellé anglais pour les conditions météo. Pour le coup, nous vous épargnerons une table de traductions… Nous nous con-

tenterons de la V.O., que nous placerons dans les attributs `alt` et `title` de l'image, à condition bien sûr de ne pas tomber sur `N/A`.

On peut être surpris par les noms pour le moins abscons des éléments et attributs : `dnam`, `tm`, `d`, `t`, `p`... Ceci dit, outre le fait qu'ils sont à peu près cohérents (respectivement *data name*, *time*, *day*, *title*, *part*), ils sont nécessaires pour un service utilisé plusieurs centaines de milliers de fois par seconde au niveau mondial.

XML est un format textuel verbeux, lourd en bande passante ; aussi allons-nous tenter de réduire la taille des données. Évidemment, pour réduire *vraiment* la taille, il aurait été bien plus malin de retirer tout formatage de la grappe (retours chariot, indentation)...

Initialisation de la page et obtention des prévisions

Nous allons donc créer notre script de façon à ce qu'il récupère les prévisions au démarrage. Nous stockerons la feuille XSLT dans une variable globale, bien entendu. Nous stockerons également les informations de prévision (code de lieu, nombre de jours) dans un objet sur lequel on évaluera un `Template` représentant l'URL à requêter.

Voici le début de notre script `client.js`.

Listing 8-16 Le début de notre script client.js

```
DEFAULT_LOCATION = 'FRXX0076';
FORECAST_DAYS = 4;
FORECAST_REFRESH = 2 * 60 * 60 * 1000;
FORECAST_TEMPLATE = new Template(
    'http://xoap.weather.com/weather/local/#{code}?dayf=#{daysAhead}&unit=m');

var gForecastParams = {
    code: DEFAULT_LOCATION,
    daysAhead: FORECAST_DAYS
};
var gForecastTimer;
var gForecastXSLT;
```

La variable `gForecastTimer` sera utilisée pour mettre à jour périodiquement (2 heures, soit `FORECAST_REFRESH` millisecondes) les prévisions.

Nous allons également nous doter d'un mécanisme de gestion pour le *timer* nécessaire à cette action périodique, et de fonctions pour afficher ou masquer l'indicateur, tout en nettoyant son texte interne ; en effet, comme pour l'exemple Amazon.fr, nous serons parfois amenés à lui ajouter un libellé temporaire.

Voici donc quelques fonctions utilitaires supplémentaires.

Listing 8-17 Quelques fonctions utilitaires

```
function cancelTimers() {
    if (gForecastTimer)
        window.clearInterval(gForecastTimer);
} // cancelTimers

function hideIndicator() {
    with ($('indicator')) {
        hide();
        update('');
    }
} // hideIndicator

function showIndicator() {
    with ($('indicator')) {
        update('');
        show();
    }
} // showIndicator

Ajax.Responders.register({
    onException: function(requester, e) {
        $('indicator').hide();
        alert(e);
    }
});

Event.observe(window, 'unload', cancelTimers);
```

La fonction `cancelTimers` n'est pas destinée à être utilisée autrement qu'en déchargement de page (fermeture du navigateur, navigation ailleurs, etc.) : nous n'y prenons donc pas spécialement la peine de remettre la variable à `0` ou `null`. Notez d'ailleurs son enregistrement sur cet événement.

Remarquez aussi le même gestionnaire d'exception Ajax que précédemment, qui cachera notre indicateur et affichera l'exception dans une boîte de message.

À présent, voyons la fonction d'extraction des prévisions.

Listing 8-18 Notre fonction d'extraction des prévisions

```
function getForecast(e) {
    if (e) ❶
        Event.stop(e);
    var url = FORECAST_TEMPLATE.evaluate(gForecastParams);
```

```
    showIndicator();
    $('results').update(''); ❷
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('indicator').update('Mise en forme&#8230;');
            var tmr = window.setTimeout(function() { ❸
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsltProcess(data, gForecastXSLT);
                $('results').update(html);
                hideIndicator();
            }, 10);
        }
    });
} // getForecast
```

L'examen initial ❶ de présence d'un argument `e` est dû au fait que nous appellerons parfois la méthode directement, alors qu'elle sera aussi invoquée lors de l'envoi de notre futur formulaire de sélection de ville. Elle n'aura donc pas toujours d'objet événement associé.

Comme dans l'exemple Amazon.fr, nous commençons par vider le conteneur de résultats ❷, qui contient peut-être la représentation de prévisions précédentes. Ainsi, l'utilisateur comprend immédiatement qu'une mise à jour a lieu, d'autant plus qu'il a peut-être remarqué l'indicateur en bas à gauche de la page.

Enfin, nous utilisons la même astuce qu'auparavant ❸ pour donner de meilleures chances à notre libellé « Mise en forme… » d'être effectivement affiché pendant la transformation XSLT.

Il ne nous reste plus qu'à initialiser la page, ce qui se fait en deux temps :

1. D'abord, nous récupérons la feuille XSLT capable de transformer un résultat XML de prévisions TWC en un fragment XHTML adapté à nos besoins.
2. Ensuite, nous lançons la demande de prévisions.

Voici le code nécessaire.

Listing 8-19 Initialisation de la page

```
function initPage() {
    new Ajax.Request('/xsl/forecast.xsl', {
        method: 'get',
        onSuccess: function(requester) {
            gForecastXSLT = xmlParse(requester.responseText);
```

```
function initPage() {
    new Ajax.Request('/xsl/forecast.xsl', {
        method: 'get',
        onSuccess: function(requester) {
            gForecastXSLT = xmlParse(requester.responseText);
            getForecast(); ❶
            gForecastTimer = window.setInterval(getForecast,
                                ➥ FORECAST_REFRESH); ❷
        }
    });
} // initPage

logging__ = false;

Event.observe(window, 'load', initPage);
```

Tout ceci doit commencer à sembler familier... On récupère la feuille XSLT, et une fois celle-ci obtenue, on la stocke dans `gForecastXSLT`, on demande ❶ les prévisions (vous voyez qu'ici, aucun objet événement n'est transmis en argument), et on déclenche le *timer* d'actualisation, toutes les 2 heures ❷.

Remarquez la fameuse variable globale `logging__` de Google AJAXSLT, que nous mettons à `false` pour éviter d'avoir un pavé de suivi des opérations XPath/XSLT en haut à droite de la page, comme dans l'exemple précédent.

Pour chipoter, on peut argumenter que puisque la requête REST est asynchrone, `initPage` récupérera la main sans doute avant qu'on ait reçu et traité les prévisions, ce qui veut dire qu'on déclenchera le *timer* une ou deux seconde(s) trop tôt. C'est exact. Cependant sur 2 heures, dans un contexte sans contraintes précises de temps, ce n'est pas vital.

Et la feuille XSLT ?

Peut-être vous êtes-vous jeté(e) sur votre page pour tester le résultat, mais n'avez abouti qu'à du vide, voire une erreur. Eh oui ! Vous avez négligé un composant critique, que nous n'avons pas encore réalisé : la feuille XSLT. La voici.

Listing 8-20 Notre feuille XSLT pour les prévisions

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
  ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="/weather"/>
    <xsl:apply-templates select="/error"/> ❶
  </xsl:template>
```

```
<xsl:template match="weather">
  <div id="weatherData"> ❷
    <xsl:apply-templates select="loc"/>
    <xsl:apply-templates select="dayf"/>
  </div>
</xsl:template>
<xsl:template match="loc">
  <p class="cityInfo">
    <xsl:value-of select="dnam"/>
    <xsl:text> @ </xsl:text>
    <span id="cityTime"><xsl:value-of select="tm"/></span> ❸
  </p>
</xsl:template>
<xsl:template match="dayf">
  <div class="forecast">
    <xsl:for-each select="day">
    <xsl:sort select="@d"/> ❹
      <div class="dayForecast">
        <span class="dayName"><xsl:value-of select="@t"/></span> ❸
        <xsl:element name="img">
          <xsl:attribute name="class">picto</xsl:attribute>
          <xsl:attribute name="src">icons/<xsl:value-of
                ➥ select="part[@p='d']/icon"/>.png</xsl:attribute>
          <xsl:if test="part[@p='d']/t != 'N/A'"> ❺
            <xsl:attribute name="alt"><xsl:value-of
                ➥ select="part[@p='d']/t"/></xsl:attribute>
            <xsl:attribute name="title"><xsl:value-of
                ➥ select="part[@p='d']/t"/></xsl:attribute>
          </xsl:if>
        </xsl:element>
        <span class="temps">
          <xsl:if test="low != 'N/A'"> ❻
            <xsl:value-of select="low"/>
          </xsl:if>
          <xsl:if test="hi != 'N/A'">
            <xsl:if test="low != 'N/A'">/</xsl:if>
            <xsl:value-of select="hi"/>
          </xsl:if>
          <xsl:if test="hi != 'N/A' or low != 'N/A'">&deg;C</xsl:if>
        </span>
      </div>
    </xsl:for-each>
  </div>
  <p class="forecastUpdateTime">
    Derni&egrave;re mise &agrave; jour des mesures :
    <span id="latestUpdate"><xsl:value-of select="lsup"/></span> ❸
  </p>
</xsl:template>
<xsl:template match="error"> ❶
  <p class="error">
```

```
      Erreur #<xsl:value-of select="err/@type"/> :
      <xsl:value-of select="err"/>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

Finalement, elle est plus légère que celle de notre exemple Amazon.fr. Nous vous recommandons tout de même de la récupérer dans l'archive des codes source du livre... Voici quelques remarques importantes :

1. Lorsqu'on a commis une erreur dans la requête, ou qu'un problème réseau ou serveur a été rencontré, on reçoit parfois une réponse contenant plutôt un élément racine `error`, dont l'élément fils `err` détaille un minimum le souci. Autant utiliser la feuille XSLT pour l'afficher en cas de pépin ! Nous n'aurons de toutes façons jamais un élément racine `weather`*et* une autre racine `error`, par définition.
2. En encadrant notre sortie dans un `div` dédié, nous isolons la portion à styler du contenant existant. Nous pouvons fournir à côté un fragment CSS dont toutes les règles sont préfixées par `#weatherData`, et être tranquilles. Ceux souhaitant plusieurs affichages utiliseraient une classe, mais nous rappellons que c'est contraire aux règles d'utilisation du service de TWC...
3. Notez les `span` avec un ID ou une classe... Outre la possibilité de styler individuellement les données au sein d'un texte environnant, ils nous seront surtout utiles pour le reformatage partiel des informations, à l'aide d'expressions rationnelles, de `gsub` et d'objets JavaScript `Date`, comme nous allons le voir plus tard.
4. Voici le fameux tri explicite des journées de prévision, que nous mentionnions plus haut.
5. Comme indiqué plus tôt, on ne définit les attributs `alt` et `title` de l'image que si le libellé n'est pas `N/A`.
6. Le traitement des `N/A` est ici un peu plus complexe, puisque nous avons deux données à gérer. Ce fragment n'affiche rien si nous avons deux `N/A`, affiche une seule température si besoin, ou les deux séparées par une barre oblique (*slash* `/`). Dès que nous avons une température, nous suffixons par `°C`.

Évidemment, il faut compléter la CSS.

Listing 8-21 Compléments de client.css

```
/* Affichage des prévisions */

#weatherData {
    margin: 1em auto;
    border: 1px solid #666;
    width: 570px;
}
```

```
p.cityInfo {
    text-align: center;
    font-weight: bold;
    font-size: smaller;
    color: navy;
    border-bottom: 1px solid silver;
    margin: 0 0 0.3em 0;
    padding: 0.1em 0;
}

#cityTime {
    color: #44f;
}

div.forecast {
    width: 552px;
    height: 180px;
    margin: 0 auto;
}

div.dayForecast {
    float: left;
    position: relative;
    width: 138px;
    height: 100%;
}

span.dayName {
    position: absolute;
    left: 0; top: 0; width: 128px;
    text-align: center;
    color: gray;
}

img.picto {
    position: absolute;
    left: 0; top: 32px;
}

span.temps {
    position: absolute;
    left: 0; bottom: 0; width: 128px;
    text-align: center;
    color: #444;
}

p.forecastUpdateTime {
    text-align: center;
    font-size: smaller;
    color: gray;
```

```
    border-top: 1px solid silver;
    margin: 0.3em 0 0 0;
    padding: 0.1em 0;
}

p.error {
    width: 60ex;
    margin: 1em auto;
    padding: 1em;
    border: 1px solid maroon;
    background: #fdd;
    color: red;
    text-align: center;
}
```

Cette fois ça y est ! Vous pouvez rafraîchir (pensez à contourner le cache), cela devrait marcher (figure 8-8).

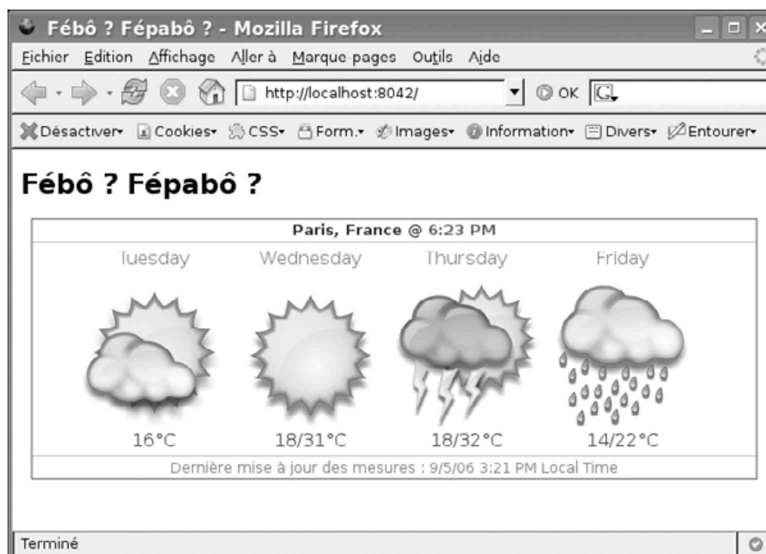


Figure 8–8
Nos prévisions à 4 jours pour Paris

Les petites touches finales

Il nous reste à apporter quelques petits ajustements qui font, pourtant, toute la différence... Si vous observez bien la figure 8-8 (ou votre écran), vous remarquerez que :

- L'heure locale est en anglais. (*damned!*)
- Les noms des jours aussi.

- Et l'heure de dernière mise à jour des mesures, n'en parlons pas !
- Qui plus est, l'heure locale « n'avance » pas comme une horloge, alors que nous souhaitions en réaliser une.

Il est temps de nous attaquer à tout cela. Commençons par modifier les textes vers des valeurs françaises (ou, dans le cas des dernières mesures, et par paresse, des valeurs *dans la langue active du navigateur*).

Pour ce faire, nous allons utiliser le même genre d'algorithme que pour notre exemple Amazon.fr, avec une fonction `adjustData`, des expressions rationnelles, des appels à la méthode `gsub` ajoutée aux `Strings` par Prototype... mais aussi quelques astuces spécifiques à cet exemple.

Voici déjà quelques « constantes » supplémentaires à ajouter en début de script.

Listing 8-22 Nouvelles « constantes » en début de script

```
DAY_NAMES = {
    'Monday': 'lundi', 'Tuesday': 'mardi',
    'Wednesday': 'mercredi', 'Thursday': 'jeudi',
    'Friday': 'vendredi', 'Saturday': 'samedi',
    'Sunday': 'dimanche'
};

RE_LOCALTIME = '(<span id="cityTime">)(.*?)(</span>)';
RE_LATESTUPDATE = '(<span id="latestUpdate">)(.*?) Local Time(</span>)';
RE_DAYNAME = '(<span class="dayName">)(.*?)(</span>)';
```

L'objet `DAY_NAMES` est une première astuce : en nommant ses propriétés d'après les noms anglais des jours, nous pourrons faire correspondre un nom français à chaque nom anglais, rien qu'en faisant `DAY_NAMES[variableDeNomAnglais]` (souvenez-vous que l'opérateur `[]` permet d'accéder aux propriétés).

Ici, nous aurions pu nous passer des apostrophes autour des noms des propriétés, mais cela n'est pas toujours vrai (certaines langues utilisent dans leurs noms de jours des caractères invalides dans un identifiant JavaScript) : autant prendre ses précautions.

Côté expressions rationnelles, le quantificateur `*?` est la version **réticente** de `*` : il signifie « prend la plus petite quantité permettant toujours de satisfaire l'ensemble de l'expression ». Dans le cas qui nous occupe, on est ainsi sûr de s'arrêter au premier `</span>` rencontré, plutôt qu'au dernier (ce qui poserait de gros problèmes). Par conséquent, nos groupes `.*?` contiennent tout ce qui se trouve entre le `>` fermant et le premier `</span>` rencontré : en somme, le contenu du `span`.

Nous aurons aussi besoin d'un nouveau *timer* pour gérer l'horloge, et d'une autre variable, un objet `Date`, pour l'heure de cette horloge. Au passage, nous ajusterons donc `cancelTimers` pour prendre en compte ce second *timer*.

```
var gLocalTime;
var gLocalTimeTimer;

function cancelTimers() {
    if (gForecastTimer)
        window.clearInterval(gForecastTimer);
    if (gLocalTimeTimer)
        window.clearInterval(gLocalTimeTimer);
} // cancelTimers
```

Il est temps d'écrire notre fameuse fonction `adjustData`. Son code est intéressant à comprendre.

Listing 8-23 Notre fonction d'ajustement de XHTML, adjustData

```
function adjustData(html) {
    // L'heure locale courante du lieu affiché
    html = html.gsub(RE_LOCALTIME, function(match) {
        gLocalTime = new Date('1/1/1970 ' + match[2]); ❶
        gLocalTime.setSeconds(new Date().getSeconds());
        return match[1] + gLocalTime.toLocaleTimeString() + match[3]; ❷
    });
    // L'heure de dernière mise à jour
    html = html.gsub(RE_LATESTUPDATE, function(match) {
        var lsup = new Date(match[2]);
        if (lsup.getFullYear() < 1980) ❸
            lsup.setFullYear(lsup.getFullYear() + 100);
        return match[1] + lsup.toLocaleString() + match[3];
    }); ❷
    // Jours de la semaine
    html = html.gsub(RE_DAYNAME, function(match) {
        return match[1] + DAY_NAMES[match[2]] + match[3];
    });
    return html;
} // adjustData
```

Voyons ensemble les quelques points saillants :

❶ Il n'est pas possible de construire un objet `Date` juste sur la base d'un texte horaire : il faut une partie date. Dans la mesure où nous ne nous intéressons cependant pas à la date (nous ne l'afficherons pas), nous en prenons une quelconque. Ici, nous avons opté pour le célèbre 1er janvier 1970 : dans la plupart des systèmes, la représentation zéro pour une date correspond au 1er janvier 1970 à

00:00:00 GMT. Remarquez l'espace en fin de texte, sans quoi la chaîne obtenue n'est plus valide (`01/01/19704:18 PM`…).
Autre point important : comme nous allons afficher les secondes (après tout, c'est une horloge détaillée…), il nous faut autre chose que les zéro secondes que va produire `Date.parse`. Quel que soit le fuseau horaire, les secondes sont les mêmes (d'accord, il existe *un* cas de figure où cela ne sera pas vrai) : nous utilisons donc les secondes locales.
Enfin, remarquez qu'on stocke l'heure de l'horloge dans la variable *globale* `gLocalTime`, que nous pourrons utiliser ensuite pour mettre à jour l'horloge.

❷ Les appels à `toLocaleTimeString()` et `toLocaleString()` demandent au navigateur de fournir une représentation textuelle complète adaptée aux conventions de la langue active. La première n'affiche que l'heure, l'autre affiche aussi la date.

❸ Les implémentations de JavaScript n'ont pour la plupart pas de « pivot » sur l'interprétation des dates. Du coup, une date à partir de 2000 exprimée sur deux chiffres va utiliser l'ancienne interprétation, et régresser de cent ans ! Nous corrigeons manuellement ce cas de figure, qui nous concerne d'ailleurs, en supposant un pivot à 1980.

Que nous reste-t-il à faire ? Tout d'abord, il va nous falloir écrire une fonction de mise à jour de l'horloge, qui sera appelée toutes les secondes.

Listing 8-24 Gestion de l'horloge

```
function updateLocalTime() {
    gLocalTime.setTime(gLocalTime.getTime() + 1000);
    $('cityTime').update(gLocalTime.toLocaleTimeString());
} // updateLocalTime
```

Ensuite, il nous faut appeler `adjustData` : pour l'instant, le code ne s'en sert pas. Et en prévision des rafraîchissements de prévisions (automatiques ou explicites, avec le formulaire que nous allons réaliser dans un instant), nous devons désactiver le *timer* d'horloge avant rafraîchissement, et le réactiver une fois le nouveau fragment XHTML affiché (et encore, à condition qu'il ne s'agisse pas d'une erreur, et qu'on dispose donc d'un élément d'ID `cityTime`).

Tout ceci a lieu dans la nouvelle version de `getForecast`.

Listing 8-25 Notre nouvelle fonction getForecast

```
function getForecast(e) {
  if (e)
    Event.stop(e);
```

```
    if (gLocalTimeTimer)
      window.clearInterval(gLocalTimeTimer);
    var url = FORECAST_TEMPLATE.evaluate(gForecastParams);
    showIndicator();
    $('results').update('');
    new Ajax.Request('/xmlProxy', {
      method: 'get',
      parameters: 'url=' + encodeURIComponent(url),
      onFailure: hideIndicator,
      onSuccess: function(requester) {
        $('indicator').update('Mise en forme&#8230;');
        var tmr = window.setTimeout(function() {
          window.clearTimeout(tmr);
          var data = xmlParse(requester.responseText);
          var html = xsltProcess(data, gForecastXSLT);
          $('results').update(adjustData(html));
          if ($('cityTime'))
            gLocalTimeTimer = window.setInterval(updateLocalTime, 1000);
          hideIndicator();
        }, 10);
      }
    });
  } // getForecast
```

Un petit rafraîchissement strict, et voici l'écran de la figure 8-9.

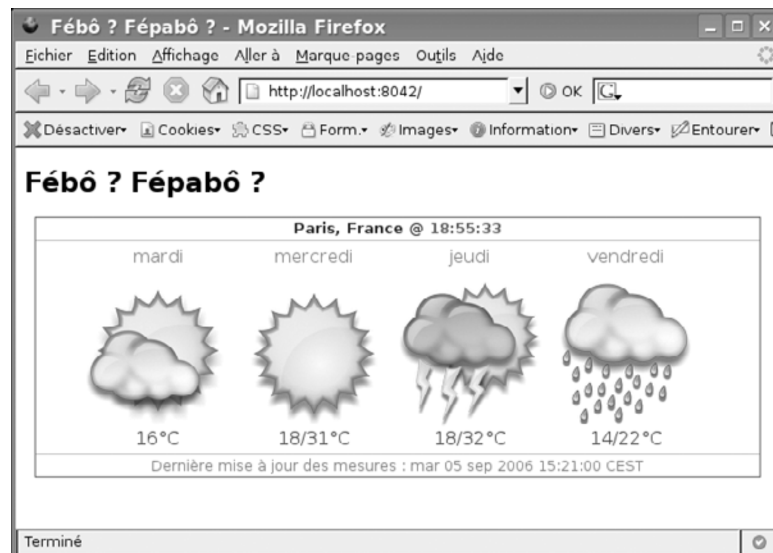


Figure 8–9
Un affichage localisé, et une horloge qui fonctionne

C'est déjà beau, je vous sens ému(e), mais ce petit code n'est rien à côté de ce qui nous attend à la prochaine section.

Rechercher un lieu

Nous allons ajouter une dernière fonctionnalité à cette page si sympathique : la possibilité de changer de lieu. Et pour cela, nous allons devoir requêter une autre ressource de l'API REST de TWC.

Cependant, cette requête exige que le nom soit assez complètement tapé pour avoir des correspondances exactes (par exemple, « Pa » ne renverra rien : pour avoir des suggestions de « Paris », il faut taper « Paris »… on a alors quatorze résultats !). On utilisera une complétion automatique de texte Ajax, pour que ce soit plus joli et plus intuitif. Cependant cette optique va nous amener à réaliser un code qui, s'il n'est pas très compliqué, va néanmoins vous remplir de fierté.

Préparons le terrain

Commençons par définir le formulaire, en début de corps de page dans `index.html`, et ajouter les styles correspondants dans `client.css`.

Listing 8-26 Le formulaire inséré dans index.html

```
<h1>Fébô ? Fépabô ?</h1>

<form id="cityForm">
  <p>
    <label for="edtCity" accesskey="V">Ville (nom anglais)</label>
    <input type="text" id="edtCity" name="city" value="Paris, France" />
    <input type="submit" id="btnDisplay" value="Afficher"
           ➥ disabled="disabled" />
    <div id="city_suggestions" class="autocomplete"></div>
  </p>
</form>

<div id="results"></div>
```

Remarquez que nous avons désactivé le bouton d'envoi, comme pour Amazon.fr : nous ne le réactiverons qu'une fois chargée la feuille XSLT (nécessaire à la transformation des résultats de recherche).

Listing 8-27 Les styles pour le formulaire, dans client.css

```
#cityForm {
    width: 34em;
    border: 1px solid gray;
    margin: 1em auto;
    padding: 0.5em;
}
```

```
#cityForm p {
    position: relative;
}

#edtCity {
    position: absolute;
    left: 10em;
    width: 17em;
}

#btnDisplay {
    position: absolute;
    left: 28em;
    width: 6em;
}

div.autocomplete {
    position: absolute;     /* Histoire d'être explicite... */
    width: 250px;           /* Sera ajusté par script.aculo.us */
    background-color: white;
    border: 1px solid #888;
    margin: 0px;
    padding: 0px;
    z-index: 42;
}

div.autocomplete ul {
    list-style-type: none;
    margin: 0px;
    padding: 0px;
}

div.autocomplete ul li {
    list-style-type: none;
    display: block;
    margin: 0;
    cursor: default;
    /* Et quelques finasseries... */
    padding: 0.1em 0.5ex;
    font-family: sans-serif;
    font-size: 90%;
    color: #444;
    height: 1.5em;
    line-height: 1.5em;
    overflow: hidden;
}

div.autocomplete ul li span.informal { (1)
    display: none;
}
```

```
/* Rappel : script.aculo.us active une classe 'selected'
   lorsqu'un élément est sélectionné */
div.autocomplete ul li.selected {
    background-color: #ffb;
}
```

Remarquez que nous utilisons exactement le même bloc de styles basiques que dans notre exemple de complétion simple au chapitre 7. Ces styles viennent d'ailleurs directement des suggestions du wiki documentaire de script.aculo.us.

Nous attirons toutefois votre attention sur la déclaration ❶. Nous avons vu que script.aculo.us ignorera tout texte contenu dans un élément de classe `informal` lorsqu'il extraira de l'élément `li` le texte à placer dans le champ de saisie. En général, nous affichons néanmoins de tels textes dans la liste des suggestions. Ici non, nous verrons pourquoi tout à l'heure.

Nous allons aussi ajouter à notre script `client.js` une « constante » pour composer l'URL de requête, ainsi qu'une variable pour recevoir la feuille XSLT dont nous aurons besoin.

Listing 8-28 Ajouts à notre script client.js

```
LOCATION_TEMPLATE = new Template(
    'http://xoap.weather.com/search/search?where=#{name}');

var gSearchXSLT;
```

Regardons à présent à quoi ressemblent notre requête et le XML de réponse. La requête est simple : l'URL http://xoap.weather.com/search/search accepte un paramètre `where` avec le texte saisi. Par exemple :

```
http://xoap.weather.com/search/search?where=paris
```

Le XML résultat est également trivial. Voici le résultat pour « Moscow ».

Listing 8-29 Résultat XML d'une recherche de lieu sur « Moscow »

```
<search ver="2.0">
  <locid="RSXX0063" type="1">Moscow, Russia</loc>
  <loc id="USAR0390" type="1">Moscow, AR</loc>
  <loc id="USID0170" type="1">Moscow, ID</loc>
  <loc id="USIA0594" type="1">Moscow, IA</loc>
  <loc id="USKS0396" type="1">Moscow, KS</loc>
  <loc id="USMI0572" type="1">Moscow, MI</loc>
  <loc id="USOH0629" type="1">Moscow, OH</loc>
  <loc id="USPA1102" type="1">Moscow, PA</loc>
```

```
  <loc id="USTN0346" type="1">Moscow, TN</loc>
  <loc id="USTX0919" type="1">Moscow, TX</loc>
  <loc id="USVT0151" type="1">Moscow, VT</loc>
</search>
```

Comme vous pouvez le constater, les américains ont décidément eu à cœur de semer des noms de villes européennes dans un état sur quatre... Il en est de même pour Florence, London... Et Paris, évidemment.

En tout cas, le XML est trivial. Le seul souci, c'est qu'il est inutilisable avec notre complétion automatique Ajax : celui-ci exige une structure `ul/li`, alors que nous avons ici du `search/loc`. Et `Ajax.Autocompleter` exploite directement le contenu récupéré. Comment faire ?

Éblouissez vos amis avec Ajax.XSLTCompleter

Nous commencerons par utiliser certaines options avancées de `Ajax.Autocompleter`, que nous n'avons guère eu l'occasion d'employer jusque-là :

- `callback`, qui permet de fournir une méthode chargée de transformer les paramètres de requête calculés par défaut ; nous l'utiliserons pour remplacer le simple nom de lieu par notre fameux paramètre `url` avec l'URL complète de requête au service REST, dont a besoin `/xmlProxy`.
- `afterUpdateElement`, qui nous donne la main une fois une sélection établie par l'utilisateur. Ici, avoir le nom de la ville intéresse l'utilisateur, mais nous, nous avons besoin du code interne (par exemple, `FRXX0076`). Il faudra le stocker dans le champ `code` de notre variable `gForecastParams`, pour que `getForecast` l'utilise dans sa requête.

Ces deux options ne résolvent toutefois pas le problème de transformation à la volée d'un format de document (le XML résultat de TWC) en un fragment XHTML `ul/li`.

Pour cela, nous allons devoir véritablement mettre les mains dans le cambouis, en créant **une sous-classe de** `Ajax.Autocompleter` ! Oui, vous avez bien lu : nous allons étendre une classe Prototype, et d'une manière non triviale.

L'idée est simple : nous allons créer un objet « héritant » de `Ajax.Autocompleter`, qui définira sa propre fonction de rappel `onComplete`, à la place de l'existante (qui existe en interne dans le prototype parent). Appelée en fin de récupération de réponse, cette méthode pourra analyser le texte de celle-ci pour produire une grappe XML, et appliquer une transformation. Il ne lui reste alors plus qu'à appeler la méthode-clé chez les objets de complétion automatique : `updateChoices`, en lui passant le fragment XHTML. Et le tour est joué ! Dans `Ajax.Autocompleter`, on appelle `updateChoices` en lui passant directement le `responseText`. Ici, nous interceptons le résultat pour le transformer.

Nous allons évidemment rendre cet objet réutilisable, en ajoutant une option `xslt` à la liste des options existantes. Cerise sur le gâteau, cette option pourra être soit du texte, soit un DOM. Dans le premier cas, on l'analysera à l'instanciation pour produire un DOM.

Le code résultat fait appel à quelques morceaux ardus de JavaScript, en particulier si vous n'avez pas bien suivi l'héritage par prototypes au chapitre 2. Toutefois, lorsque vous aurez compris ce code, vous serez bien avancé(e) sur le chemin qui mène au statut de gourou JavaScript !

Voici notre nouvel objet, à ajouter dans `client.js`.

Listing 8-30 Notre objet Ajax.Autocompleter

```
Ajax.XSLTCompleter = Class.create();
Object.extend(Ajax.XSLTCompleter.prototype,
Ajax.Autocompleter.prototype);
Object.extend(Ajax.XSLTCompleter.prototype, {
  initialize: function() { ❶
    Ajax.Autocompleter.prototype.initialize.apply(this, arguments); ❷
    this.options.onComplete = this.onComplete.bind(this);
    var options = arguments[3] || {}
    if ("string" == typeof this.options.xslt)
      this.options.xslt = xmlParse(this.options.xslt);
  },

  onComplete: function(request) {
    var data = xmlParse(request.responseText);
    var html = xsltProcess(data, this.options.xslt);
    this.updateChoices(html); ❸
  }
});
```

Et voilà le travail ! Quelques précisions tout de même :

❶ Dans les objets Prototype (les objets obtenus en appelant `Class.create()`), le « constructeur » s'appelle `initialize`.

❷ Cette ligne revient à appeler la version héritée du constructeur, mais bien sur l'objet courant. C'est un peu l'équivalent d'un `super(…)` en Java (auquel on repasserait explicitement tous les arguments), ou d'un `inherited;` en Delphi.

❸ C'est cet appel qui va mettre à jour la liste des suggestions et l'afficher automatiquement.

Cet objet étant prêt, il nous reste à disposer d'une feuille de styles, et à utiliser tant l'objet que la feuille. Voici déjà le document, `search.xsl`, à mettre dans `docroot/xsl`.

Listing 8-31 Notre feuille xsl/search.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
        ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="/search"/>
  </xsl:template>
  <xsl:template match="search">
    <ul>
      <xsl:apply-templates select="loc"/>
    </ul>
  </xsl:template>
  <xsl:template match="loc">
    <li>
      <span class="informal"><xsl:value-of select="@id"/></span> ❶
      <xsl:value-of select="."/>
    </li>
  </xsl:template>
</xsl:stylesheet>
```

Vous vous demandez peut-être pourquoi nous avons utilisé un `span` de classe `informal` (que la CSS cache, qui plus est !), plutôt que l'attribut `value` du `li`, qui semble l'emplacement naturel pour le code.

Nous aussi.

Nous avons découvert que Firefox (et peut-être d'autres navigateurs) avait un défaut dans sa propriété `innerHTML`, qui retirait le début de certains codes lorsque ceux-ci étaient stockés dans l'attribut `value` des `li` ! En revanche, placés dans le `span`, aucun problème... Voilà pourquoi nous utilisons ce moyen quelque peu détourné.

Brancher les composants ensemble

C'est maintenant la dernière ligne droite de notre exemple : nous allons insérer du code supplémentaire au début de notre fonction `initPage`. Ce code a trois fonctions :

1. Associer `getForecast` à l'envoi du formulaire, comme cela a été prévu dès le départ.
2. Requêter la feuille XSLT
3. Une fois celle-ci obtenue, initialiser la complétion automatique sur le champ du formulaire, avec quelques paramètres spéciaux...

Voici donc le nouveau code, figurant au tout début de `initPage` (nous avons laissé les lignes alentour pour vous aider à vous repérer).

Listing 8-32 Notre initialisation finalisée

```
function initPage() {
  Event.observe('cityForm', 'submit', getForecast); ❶
  new Ajax.Request('/xsl/search.xsl', { ❷
    method: 'get',
    onSuccess: function(requester) {
      gSearchXSLT = xmlParse(requester.responseText);
      new Ajax.XSLTCompleter('edtCity', 'city_suggestions',
             ➥ '/xmlProxy', { ❸
        method: 'get',
        paramName: 'foo', ❹
        indicator: 'indicator', ❺
        minChars: 2, ❻
        xslt: gSearchXSLT, ❼
        callback: function(e, entry) { ❽
          var url = LOCATION_TEMPLATE.evaluate(
                         ➥ { name: entry.substring(4) });
          return 'url=' + encodeURIComponent(url);
        },
        afterUpdateElement: function(e, elt) {
         gForecastParams.code = elt.firstChild.firstChild.nodeValue; ❾
        }
      });
      $('btnDisplay').disabled = false; ❿
    }
  });
  new Ajax.Request('/xsl/forecast.xsl', {
```

Vu la richesse de ce code, nous vous gratifions d'un bon nombre de remarques :

❶ Nous associons, évidemment, `getForecast` à l'envoi du formulaire. Souvenez-vous du `Event.stop` au début de `getForecast` : cela empêchera l'envoi normal, au profit d'une recherche Ajax. Celle-ci utilisera `gForecastParams.code`, qui aura été mis à jour par la fonction de rappel `afterUpdateElement` de la complétion automatique.

❷ Voilà notre chargement de feuille XSLT pour transformer les résultats de recherche en fragment compatible (`ul/li`) avec `Autocompleter.Base`, le mécanisme générique de complétion automatique.

❸ Évidemment, nous utilisons à nouveau notre `/xmlProxy` à tout faire...

❹ Nous pourrions nous en passer, mais comme le seul paramètre que nous souhaitions est en fait le paramètre `url` que va synthétiser notre fonction de rappel `callback`, autant bien indiquer que le paramètre d'origine ne sert à rien côté serveur (il ne sera même pas envoyé, car `callback` ne le préserve pas).

❺ Facilité toujours : vous souveniez-vous que l'option `indicator` affiche et masque toute seule notre indicateur ? Puisqu'ici nous ne nous soucions pas de jouer sur son libellé, nous pouvons utiliser cette option.

❻ TWC envoie une erreur si nous effectuons une recherche avec un nom d'un seul caractère pour le paramètre `where`. Il est vrai que ce n'est pas très utile. Disons 2 au minimum.

❼ Hourra ! Voici notre option personnalisée ! Nous lui transmettons directement notre variable globale qui contient la feuille XSLT à utiliser. Comme le code courant s'exécute depuis le `onSuccess` du chargement de cette XSLT, on sait que la variable est prête.

❽ La fonction de rappel `callback` reçoit le champ de saisie et le paramètre correspondant à la saisie en cours, prévu pour envoi côté serveur. Ici, c'est donc par exemple `'foo=blah'` (pour une saisie de `blah`, et parce que nous avons dit `paramName: 'foo'`).
Nous éliminons les 4 premiers caractères pour récupérer le texte encodé, que nous injectons dans notre modèle d'URL. Nous encodons le tout et nous préfixons par `'url='`, le paramètre attendu par `/xmlProxy`.

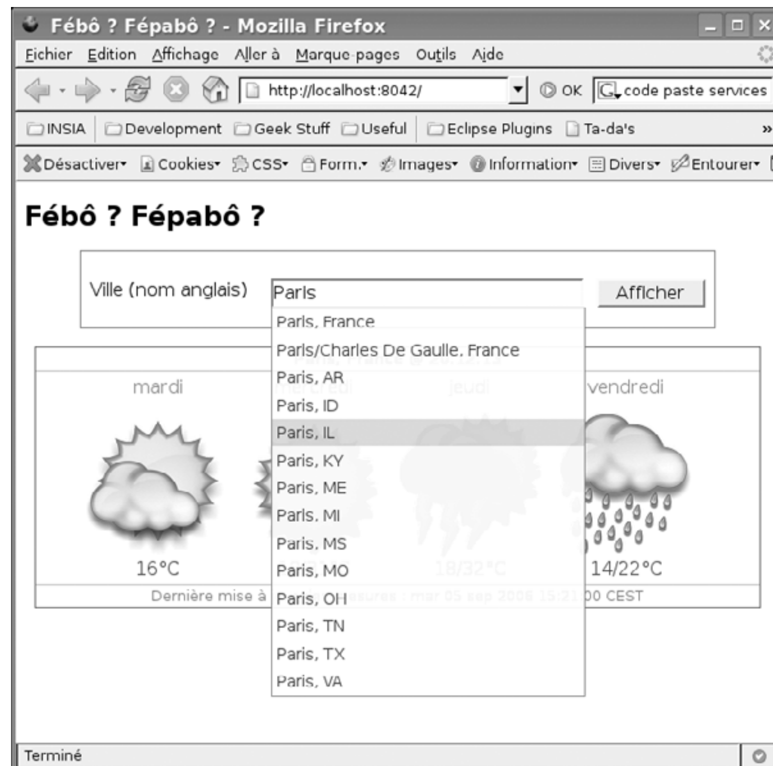


Figure 8–10
Les lieux possibles pour « Paris »

9 Relisez la feuille XSLT : chaque `li` contient d'abord un premier élément fils `span`, qui contient un seul élément fils de type texte. Pour accéder à la valeur de ce texte, on a donc `elt` (l'élément `li`), un premier `firstChild` (le `span`), un second (le nœud texte dans le `span`), et un `nodeValue` (le texte lui-même).

10 N'oublions pas, après tout ça, de finir de réagir au chargement de la feuille XSLT en activant le bouton du formulaire !

Cette fois vous y êtes : vous pouvez faire un rafraîchissement strict, et taper autre chose dans la zone, par exemple « Paris ». Vous devriez voir rapidement apparaître une liste de possibilités (figure 8-10).

Remarquez « Paris/Charles de Gaulle, France » : les services météo traitent généralement à part les aéroports. Choisissons par exemple « Paris, IL » (Illinois, USA), et validons le formulaire (sous Opera 9 et Safari 2, il est validé directement), ce qui donne l'écran de la figure 8-11.



Figure 8–11
Le temps dans l'Illinois

Notez l'heure locale. L'Illinois, c'est 7 heures avant la France...

Allez, il est l'heure d'appeler les collègues et de montrer fièrement vos prouesses...

Gérer des images chez Flickr

Flickr, récemment acquis par Yahoo!, est le site de référence pour publier, partager, cataloguer et diffuser des photos en ligne. Fort d'une direction technique de haut vol, le site propose une API très complète, accessible en REST, en XML-RPC ou en SOAP (rien que dans l'ordre de ces formats, qui est celui figurant partout sur le site, on sent que la couche technique va dans le bon sens...).

Cette API est merveilleusement documentée, de façon très claire et détaillée. Le point d'entrée de la documentation est http://www.flickr.com/services/api/ (tout simplement). Chaque méthode est décrite avec son synopsis, le détail de ses arguments, ses contraintes en authentification, un exemple du contenu de réponse, et la liste détaillée des codes d'erreur avec leurs causes !

C'est déjà fabuleux, mais on trouve en plus à chaque fois un lien sur l'explorateur API, qui permet de tester immédiatement, interactivement, un appel à la méthode (pour peu qu'on soit connecté et qu'on dispose d'une clé API). Un vrai bonheur...

Par ailleurs, de nombreuses bibliothèques pour cette API sont disponibles, pour la plupart des langages et plates-formes (citons notamment, par ordre alphabétique : ActionScript, Delphi, Java, .NET, PHP, Python et Ruby).

Dans ce dernier exemple, nous allons afficher un jeu de photos publié par un utilisateur de Flickr (et pas des moindres : Tristan Nitot, le président de Mozilla Europe en personne), afficher les vignettes, et en cliquant dessus, afficher la photo plus grande, en tant que lien vers la taille originale, ainsi que les commentaires de la photo.

Nous n'utiliserons en revanche pas de méthodes modifiantes (ajout de commentaire, etc.) car elles nécessitent un mécanisme d'authentification mal adapté à nos conditions de test. Ce n'est pas tant qu'il faille calculer des *hashes* MD5 (des bibliothèques JavaScript font cela très vite), mais qu'il faille fournir à Flickr une URL *accessible sur le Web*, connectée à notre application... Cela ferait plusieurs allers-retours consécutifs, et deviendrait vite trop épais pour ce chapitre.

Obtenir une clé API chez Flickr

Il faut d'abord avoir un compte chez Flickr, ce qui signifie aujourd'hui un compte chez Yahoo!. Si vous n'avez pas de compte, créez-en un sur http://www.flickr.com/signup :

1. Choisissez le lien Sign Up sur la droite.
2. Saisissez les champs nécessaires. Attention, le code de confirmation est sensible aux majuscules/minuscules. Utilisez une adresse électronique valide, car vous y recevrez un lien d'activation de compte.

3 Récupérez votre courriel de bienvenue et suivez son lien d'activation.
4 Choisissez un nom de compte Flickr, qui peut être différent de votre nom de compte Yahoo!
5 Activez le bouton Sign In.

À présent que vous êtes connecté(e), demandez une clé en allant sur http://www.flickr.com/services/api/key.gne :

1 Vérifiez votre nom et votre adresse électronique.
2 Si vous demandez ici une clé pour une utilisation commerciale, cochez le bouton radio correspondant. Attention toutefois, le service dans un contexte commercial est généralement payant...
3 Décrivez rapidement l'utilisation que vous allez faire de la clé (par exemple, « I'm going to use it to explore your services while reading a book that uses Flickr REST examples ».
4 Cochez la case I agree to Flickr APIs Terms of Use.
5 Activez le bouton Apply.
6 Et voilà ! Votre clé s'affiche. Copiez-collez-la en lieu sûr.

Prenez le temps de lire le contrat d'utilisation de l'API. Il s'agit surtout de respecter les règles en vigueur sur Flickr, de respecter la propriété intellectuelle, et de réagir avec diligence aux demandes des propriétaires de photos que vous exposez à travers votre application. La page décrivant les conditions d'utilisation est plutôt claire, ne manquez pas de la parcourir.

Format général des requêtes et réponses REST chez Flickr

Vous pouvez vérifier que votre clé est active en utilisant la méthode (l'opération, si vous préférez) `flickr.test.echo`, comme ceci, en précisant évidemment votre clé réelle :

```
http://api.flickr.com/services/rest/
?method=flickr.test.echo&opinion=PasMalCeBouquin&api_key=VOTRE_CLÉ_ICI
```

Vous devez obtenir un résultat XML comme celui-ci :

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<method>flickr.test.echo</method>
<opinion>PasMalCeBouquin</opinion>
<api_key>VOTRE_CLÉ_ICI</api_key>
</rsp>
```

D'une manière générale, une requête REST Flickr utilise toujours le point d'accès suivant :

```
http://api.flickr.com/services/rest/
```

L'opération et le type de ressource voulus sont encodés dans la **méthode**, un paramètre `method` dont le nom utilise une syntaxe hiérarchique à points, comme les méthodes `flickr.blogs.getList`, `flickr.groups.browse` ou `flickr.photos.addTags`.

On doit obligatoirement préciser la clé API à travers le paramètre `api_key`. Le reste dépend de la méthode. Flickr n'utilise que les verbes HTTP `GET` et `POST`, ce qui n'est pas toujours sémantiquement approprié, mais ce n'est pas très grave.

Une réponse REST Flickr a deux formats possibles. Soit l'appel a réussi, et on obtient ce type de contenu.

Listing 8-33 Une réponse REST Flickr en cas de succès

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
Contenu spécifique à la méthode ici
</rsp>
```

Soit l'appel a échoué, et on obtient ce type de contenu.

Listing 8-34 Une réponse REST Flickr en cas d'échec

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="fail">
    <err code="code-erreur" msg="message-erreur" />
</rsp>
```

Il est donc facile de détecter un problème rien qu'avec l'attribut `stat` de l'élément racine `rsp` (chemin XPath : `/rsp/@stat`).

Préparer le terrain

Commencez par créer un répertoire de travail `flickr`, et copiez-y le `serveur.rb` d'un des deux exemples précédents. Recréez aussi `docroot`, son sous-répertoire `ajaxslt` et un sous-répertoire `xsl` vide. Vous pouvez aussi préparer des squelettes pour `index.html`, `client.css`, `client.js`, et les *quatre* feuilles XSLT que nous allons utiliser : `photoset.xsl`, `photoset_owner.xsl`, `photoset_photos.xsl` et `photo.xsl`.

Notre page étant générée presque entièrement suite à des requêtes Ajax, nous avons un fichier `index.html` au corps plutôt simple.

Listing 8-35 Notre page, index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Les jeux de photo de Tristan&#8230;</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
            ➥ src="scriptaculous.js?load=effects,dragdrop"></script>
    <script type="text/javascript" src="ajaxslt/misc.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt/xpath.js"></script>
    <script type="text/javascript" src="ajaxslt/xslt.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<div id="indicator" style="display: none;"></div>

<div id="results"></div>

<div id="photo" style="position: absolute; display: none">
    <span id="photo-closer">[x]</span>
    <div id="photo-contents"></div>
</div>

</body>
</html>
```

Remarquez le chargement de `dragdrop.js`, qu'il vous faudra donc récupérer dans script.aculo.us (en revanche, nous n'utilisons plus `controls.js`). Le corps de la page sera dans `results`, tandis que `photo` est un conteneur destiné à afficher, en surplomb du contenu normal, une photo individuelle. Une pseudo-case de fermeture figurera en haut à droite.

Voyons à présent une première version de la feuille de styles.

Listing 8.36 Le début de notre feuille client.css

```
* {
    font-size: 1em;
}
```

```
body {
    font-family: sans-serif;
}

#indicator {
    position: absolute;
    left: 10px; top: 10px; height: 16px; width: 20em;
    font-size: 14px; line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
    color: gray;
}

#results {
    margin-top: 36px;
}
```

Rien de bien méchant. Remarquez que cette fois, on positionne l'indicateur en haut de la page, toujours afin qu'il ne sursaute pas quand on modifie le contenu...

Notre script `client.js` reprend par ailleurs quelques fragments désormais classiques, que nous compléterons par la suite.

Listing 8-37 Squelette initial de notre client.js

```
function initPage() {
    new Draggable('photo', {});
} // initPage

Ajax.Responders.register({
    onException: function(requester, e) {
        $('indicator').hide();
        alert(e);
    }
});

logging__ = false;

Event.observe(window, 'load', initPage);
```

Remarquez qu'on peut déjà définir le conteneur photo comme étant déplaçable par glissement. L'argument d'options vide est là pour corriger un très léger souci (pas d'opacité partielle au glissement) dans script.aculo.us 1.6.3-4, qui sera corrigé en 1.6.5.

Chargement centralisé parallèle des feuilles XSLT

Nous commençons à avoir de nombreuses feuilles à charger. Le code que nous avions écrit jusqu'ici les chargeait séquentiellement, et au prix d'imbrications parfois complexes de `new Ajax.Request`... Pour éviter cela, nous pouvons écrire un petit objet apte à charger autant de feuilles XSLT qu'on le souhaite, en parallèle qui plus est.

Cet objet pourrait avoir un *hash* nommé `sheets`, dont les clés seraient les noms de ses propriétés finales (qui contiendront le DOM des feuilles), et les valeurs seraient les « noms racines » des fichiers. On supposerait en effet que toutes les feuilles serait dans le même répertoire, et aurait la même extension. Cet objet pourrait charger ces feuilles en parallèle, exposer le DOM obtenu pour chacune sous forme de propriété, et prévenir une fonction de rappel quand toutes les feuilles ont été chargées.

Tout ceci semble complexe, mais en réalité, cela n'utilise rien de très difficile. Regardez l'objet anonyme suivant, que nous allons placer au début de notre fichier de script `client.js`.

Listing 8-38 Notre chargeur parallèle de feuilles XSLT

```
var gXSLT = {
    _loaded: false, ❶

    sheets: { ❷
        photoset: 'photoset',
        owner: 'photoset_owner',
        photos: 'photoset_photos',
        photo: 'photo'
    },

    load: function(onComplete, expose) { ❸
        if (this._loaded)
            return;
        this._exposeSheets = false !== expose; ❹
        this._onComplete = onComplete;
        this._loadCount = 0;
        this._sheetCount = $A($H(this.sheets).keys()).length; ❺
        for (var s in this.sheets)
            this._loadSheet(s);
    },

    _loadSheet: function(sheet) {
        new Ajax.Request('/xsl/' + this.sheets[sheet] + '.xsl', {
            method: 'get',
            onSuccess: this._sheetLoaded.bind(this, sheet)
        });
    },
```

```
    _sheetLoaded: function(sheet, requester) {
        this.sheets[sheet] = xmlParse(requester.responseText);
        if (this._exposeSheets) {
            if (undefined !== this[sheet])
                alert('Cannot expose sheet ' + sheet +
                    ➥ ': property exists.');
            this[sheet] = this.sheets[sheet]; ❻
        }
        if (++this._loadCount >= this._sheetCount) {
            this._loaded = true;
            (this._onComplete || Prototype.EmptyFunction)(); ❼
        }
    }
}; // gXSLT
```

Voyons certains fragments de plus près...

❶ Ce drapeau nous permettra d'ignorer des appels multiples à `load`.

❷ C'est là qu'on définit les feuilles à charger. Nous avons ici, par exemple, une feuille de nom racine `photoset_photos` dont le DOM devra être exposé dans une propriété `photos`.

❸ C'est cette fonction qui est destinée à être appelée de l'extérieur : elle déclenche le chargement parallèle. On lui transmet une fonction de rappel (optionnelle), et éventuellement un drapeau indiquant si on doit exposer les DOM obtenus sous forme de propriétés de l'objet `gXSLT`.

❹ En utilisant la différence stricte `!==`, nous obtenons `true` dans tout autre cas que `false` spécifiquement (donc `null` ou `undefined` donnent `true`), ce qui fait que `true` est la valeur par défaut.

❺ Petite astuce pour obtenir le nombre de propriétés dans `sheets`, donc de feuilles à charger : on en fait un `Hash`, on prend `keys()`, on convertit en tableau, et on appelle `length`.

❻ Voilà comment exposer le DOM sous forme de propriété directe (`gXSLT.photo` plutôt que `gXSLT.sheets.photo`, qui marche aussi, ceci dit). Remarquez qu'on hurle si une telle propriété existe déjà.

❼ Voici un code typique de Prototype ! Cela nous évite un `if` pour tester qu'on nous a bien transmis une fonction de rappel. C'est la raison d'être de `Prototype.EmptyFunction`.

Obtenir les informations du jeu de photos

Il est temps à présent d'attaquer le requêtage. Nous allons commencer par récupérer les informations du jeu de photos qui nous intéresse. Toutes les requêtes Flickr ayant la même structure de base, nous allons utiliser quelques « constantes » de bon aloi.

Listing 8-39 Définition du requêtage

```
FLICKR_API_KEY = 'VOTRE_CLÉ_ICI';
FLICKR_ENDPOINT = 'http://api.flickr.com/services/rest/';
FLICKR_TEMPLATE = new Template(
    FLICKR_ENDPOINT + '?method=flickr.#{method}&api_key=' +
    FLICKR_API_KEY + '#{extraArgs}');

PHOTOSET_ID = 1603279; // Ou 1583292...
```

Comme toutes les méthodes Flickr commencent par « `flickr.` », nous allons en dispenser le code qui suit, en utilisant statiquement ce préfixe. Remarquez que nous pourrons préciser les arguments spécifiques à la méthode à l'aide d'une propriété `extraArgs`. Nous devrons donc passer à `FLICKR_TEMPLATE` un objet avec deux propriétés : `method` et `extraArgs`.

Penchons-nous donc sur les informations inhérentes au jeu de photos. Nous sommes intéressés par son titre, sa description, son nombre de photos, sa photo représentative (une des photos du jeu, mise en avant) et son propriétaire. Ce dernier point nécessitera d'ailleurs une deuxième requête, car au niveau du jeu de photos (le *photoset*), nous n'avons que l'identifiant (NSID) du propriétaire.

Voici le résultat XML que Flickr envoie pour le jeu 1603279.

Listing 8-40 La réponse REST de Flickr pour un jeu de photos

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photoset id="1603279" owner="19663157@N00" primary="73723635"
          ➥ secret="a9f7246444" server="20" photos="46">
    <title>Portraits of Mozillians</title>
    <description>Taken during lunch time on Dec. 8th, 2005
</description>
</photoset>
</rsp>
```

Que voit-on dans ce résultat ?

- L'élément racine est `photoset`.
- Les informations nécessaires à l'identification de sa photo représentative sur les serveurs de photos Flickr (`static.flickr.com`) sont les attributs `server`, `primary` et `secret`. Nous expliquerons cela en détail plus loin.
- Le nombre de photos est dans l'attribut `photos`.
- L'identifiant du propriétaire est dans l'attribut `owner`.
- Le titre et la description sont dans les éléments fils.

Cette grappe est plutôt simple. Pour revenir sur les attributs d'identification de la photo, il faut savoir que chez Flickr, une photo dispose de plusieurs versions de son fichier, en différentes tailles. Ces versions sont mises à disposition sur toute une ferme de serveurs de ressources statiques, accessibles *via*`static.flickr.com`. L'URL d'une photo est composée comme suit :

```
http://static.flickr.com/[serveur]/[id]_[secret][_taille].[format]
```

L'ID d'une photo correspond ici à l'attribut `primary` du `photoset`. La partie `secret` est là pour obliger les systèmes clients à requêter l'API afin d'obtenir les informations nécessaires, diminuant ainsi considérablement les risques de saturation, d'utilisation non autorisée, et d'attaque en déni de service.

La taille peut être l'une des valeurs suivantes :

- `_s` (*square*) pour la vignette carrée (destinée à un affichage en masse, comme dans la page d'un jeu de photos, ou dans notre propre page pour ce chapitre) ;
- `_t` (*thumbnail*) pour la vignette ;
- `_m` (*medium*) pour la petite taille (environ 50 %) ;
- rien pour la taille moyenne, destinée à l'affichage par défaut ;
- `_o` (*original*) pour la photo originale.

Une photo n'a pas forcément toutes ces variantes disponibles. On peut demander la liste des tailles d'une photo (dimensions, URL, etc.) avec la méthode `flickr.photos.getSizes`.

Enfin, le format de toutes les versions dérivées est `jpg`. La version originale a un format précis, qu'on obtient en demandant les informations pour la photo en question.

Pour notre en-tête de jeu de photos, nous utiliserons la petite taille de la photo représentative. Voici la feuille `xsl/photoset.xsl`, qui transforme cette grappe en ce qui nous intéresse.

Listing 8-41 La feuille XSLT xsl/photoset.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
        ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="/rsp/photoset"/>
    <xsl:apply-templates select="/rsp/err"/>
  </xsl:template>
```

```
  <xsl:template match="photoset">
    <div id="photoset">
      <xsl:element name="img">
        <xsl:attribute name="id">
          <xsl:value-of select="@id"/>
        </xsl:attribute>
        <xsl:attribute name="src"> ❶
          http://static.flickr.com/<xsl:value-of
                 ➥ select="@server"/>/<xsl:value-of select="@primary"/>
                 ➥ _<xsl:value-of select="@secret"/>_m.jpg
        </xsl:attribute> ❷
      </xsl:element>
      <h1><xsl:value-of select="title"/></h1>
      <p class="description"><xsl:value-of select="description"/></p>
      <p class="photoCount">
        <xsl:value-of select="@photos"/>
        photo<xsl:if test="@photos &gt; 1">s</xsl:if> ❸
        de
        <span id="photoset-owner"></span> ❹
      </p>
    </td>
  </xsl:template>
  <xsl:template match="err">
    <p class="error">
      Erreur n&deg; <xsl:value-of select="@code"/> :
      <xsl:value-of select="@msg"/>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

Là aussi, nous prenons la peine de fournir une représentation en cas d'erreur. Notez la composition de l'attribut `src` pour l'image, de la ligne ❶ à la ligne ❷ : elle suit notre schéma. Nous sommes obligés de « coller » les composants pour éviter les espaces dans l'URL, qui pourraient troubler Flickr.

Remarquez aussi qu'en ❸, nous avons pris la peine de gérer le pluriel. Quant au `span` en ❹, il est vide car nous n'avons pas encore le nom, ni l'URL de profil, du propriétaire : pour les avoir, il nous faudra une deuxième requête, que nous ferons plus loin. En mettant un ID au `span`, nous pourrons mettre à jour son contenu plus tard.

Avant d'aller plus loin, pensez à réaliser un squelette XSLT valide pour les trois autres fichiers XSL référencés par `gXSLT.sheets` : sans eux, nous aurons une erreur de traitement, et notre fonction de rappel, qui doit requêter les informations du jeu de photos, ne sera jamais appelée.

Ceci étant assuré, nous allons pouvoir réaliser la fonction `getPhotoset`, qui va chercher un jeu de photos dont on lui donne l'identifiant.

Listing 8-42 Notre fonction getPhotoset

```
function getPhotoset(id) {
    $('results').update('');
    var url = FLICKR_TEMPLATE.evaluate({ ❶
        method: 'photosets.getInfo',
        extraArgs: '&photoset_id=' + id
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var ctx = new ExprContext(data); ❷
            var ownerNSID = xpathEval('/rsp/photoset/@owner',
                                      ➥ ctx).stringValue();
            var html = xsltProcess(data, gXSLT.photoset); ❸
            $('results').update(html);
            // Ici, il faudra aller chercher plus d'infos...
        }
    });
} // getPhotoset
```

Cette fonction n'a rien de révolutionnaire. Remarquez tout de même en ❶ le mécanisme que nous utiliserons à chaque fois pour construire un appel REST à Flickr, et en ❷, l'extraction manuelle du NSID de l'utilisateur Flickr propriétaire du jeu de photos. Nous allons en avoir besoin, là où se trouve pour l'instant un commentaire, pour aller chercher les informations complémentaires sur le propriétaire, afin de les afficher dans `photoset-owner`. Remarquez enfin la transformation, qui va chercher ❸ la feuille dans notre propriété exposée `gXSLT.photoset`.

Notre fonction étant prête, il ne nous reste plus qu'à compléter l'initialisation de notre page.

Listing 8-43 Une initialisation un peu plus complète

```
function initPage() {
    gXSLT.load(function() {
        getPhotoset(PHOTOSET_ID);
    });
    new Draggable('photo', {});
} // initPage
```

Voilà : on charge les feuilles XSLT (c'est pourquoi nous vous avons demandé de réaliser des squelettes valides pour les trois autres), et une fois qu'elles sont toutes chargées et interprétées avec succès, on appelle la fonction de rappel anonyme ci-dessus, qui appelle `getPhotoset` en lui passant notre constante `PHOTOSET_ID`.

Voyons déjà où cela nous mène (figure 8-12).

Figure 8–12
La récupération
des premières informations



Cela manque de cachet, tout de même. Sans doute pouvons-nous améliorer les choses en ajoutant quelques règles CSS.

Listing 8-44 Ajouts dans client.css pour l'en-tête du jeu de photos

```
#photoset h1 {
    font-family: "Bistream Vera Serif", serif;
    color: #444;
    font-size: 150%;
}

#photoset .description {
    color: #777;
}
```

```
#photoset img {
    float: left;
    margin: 0 1em 1em 0;
}

#photoset .photoCount {
    font-size: smaller;
    color: #444;
}
```

Notez que l'image flotte à gauche. Il faudra que l'élément suivant, que nous ajouterons tout à l'heure, soit stylé en `clear: both`. Voyons le nouveau look de notre en-tête (figure 8-13).



Figure 8–13
L'en-tête avec des règles CSS

Voilà qui n'est déjà pas mal ! Évidemment, il manque quelque chose après le « 46 photos de ». Pour cela, il nous faut simplement une nouvelle requête, afin d'obtenir des informations sur l'utilisateur. La méthode `flickr.people.getInfo` renvoie un résultat du type suivant.

Listing 8-45 Résultat XML d'une requête d'informations utilisateur

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<person id="19663157@N00" nsid="19663157@N00" isadmin="0"
        ➥ ispro="1" iconserver="13">
```

```
    <username>nitot</username>
    <realname>Tristan Nitot</realname>
    <mbox_sha1sum>4f24c004c2d5c0cfcfed422c6afe26d806616919
    </mbox_sha1sum>
    <location>Paris, France</location>
    <photosurl>http://www.flickr.com/photos/nitot/</photosurl>
    <profileurl>http://www.flickr.com/people/nitot/</profileurl>
    <mobileurl>http://www.flickr.com/mob/photostream.gne?id=627752
    </mobileurl>
    <photos>
        <firstdatetaken>2005-01-12 16:37:44</firstdatetaken>
        <firstdate>1115996869</firstdate>
        <count>469</count>
    </photos>
</person>
</rsp>
```

Remarquez que Tristan est prolifique (près de 500 photos). Nous sommes particulièrement intéressés par son nom complet et l'URL de son profil utilisateur, sur laquelle nous comptons lier. Là aussi, une petite feuille XSLT, `photoset_owner.xsl`, va s'occuper de tout.

Listing 8-46 La feuille xsl/photoset_owner.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
                ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/person"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="person">
        <xsl:element name="a">
            <xsl:attribute name="href">
                <xsl:value-of select="profileurl"/>
            </xsl:attribute>
            <xsl:value-of select="realname"/>
        </xsl:element>
    </xsl:template>
    <xsl:template match="err">
        <span class="error"> ❶
            Erreur n&deg; <xsl:value-of select="@code"/> :
            <xsl:value-of select="@msg"/>
        </span>
    </xsl:template>
</xsl:stylesheet>
```

C'est sans doute la feuille XSLT la plus simple du chapitre. Nous gérons tout de même les erreurs, mais comme vous pouvez le remarquer en ❶, nous utilisons ici un `span` au lieu d'un `p`. En effet, le résultat de cette feuille est censé apparaître dans l'élément `photoset-owner`, qui est lui-même un `span`. Et vous savez certainement que XHTML interdit les éléments de type bloc dans des éléments de type en ligne (ce qui est compréhensible).

Il nous reste à charger cette information juste après avoir inséré le fragment XHTML du jeu de photos dans notre DOM global. Faisons d'abord une fonction dédiée.

Listing 8-47 La fonction getOwnerInfo

```
function getOwnerInfo(nsid) {
    var url = FLICKR_TEMPLATE.evaluate({
        method: 'people.getInfo',
        extraArgs: '&user_id=' + nsid
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var html = xsltProcess(data, gXSLT.owner);
            $('photoset-owner').update(html);
        }
    });
} // getOwnerInfo
```

Et ajoutons un appel après insertion du fragment conteneur, dans `getPhotoset`.

Listing 8-48 Insertion de l'appel supplémentaire dans getPhotoset

```
function getPhotoset(id) {
    …
        onSuccess: function(requester) {
            …
            var ownerNSID = xpathEval('/rsp/photoset/@owner',
                                      ➥ ctx).stringValue();
            …
            getOwnerInfo(ownerNSID);
        }
    });
} // getPhotoset
```

Observons le résultat (figure 8-14).



Figure 8–14
Les informations complètes pour le jeu de photos

Remarquez, dans la barre d'état, l'URL cible du lien : c'est bien l'URL de la page de profil de l'utilisateur.

Récupérer les photos du jeu

À présent, il nous faut les photos du jeu de photos. Pour cela, nous allons utiliser la méthode `flickr.photosets.getPhotos`. Voici un fragment du résultat.

Listing 8-49 Fragment de résultat XML pour flickr.photosets.getPhotos

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photoset id="1603279" primary="73723635">
    <photo id="73723882" secret="015c203114" server="20"
          ➥ title="Peter Van der Beken, aka peterv" isprimary="0" />
    <photo id="73723841" secret="1fde5f4ce5" server="35"
          ➥ title="Brendan Eich" isprimary="0" />
    …
</photoset>
</rsp>
```

L'objectif est de produire un tableau de vignettes carrées, avec le titre de chaque photo dans les attributs `alt` et `title` des images. Le tableau ferait 10 vignettes de large.

Réaliser ce partitionnement en XSLT est vite atroce, aussi nous allons opter pour une astuce toute simple : nous allons générer une unique ligne avec toutes les vignettes, et utiliser quelques petits remplacements et une ou deux expressions rationnelles simples, pour transformer cela en série de lignes (`tr`).

Voici déjà la feuille XSLT qui va transformer le contenu XML.

Listing 8-50 La feuille xsl/photoset_photos.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
                ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/photoset"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="photoset">
        <table id="photoset-photos">
            <tbody>
                <tr>
                    <xsl:apply-templates select="photo"/>
                </tr>
            </tbody>
        </table>
    </xsl:template>
    <xsl:template match="photo">
        <td>
            <xsl:element name="img">
                <xsl:attribute name="id">
                    <xsl:value-of select="@id"/>
                </xsl:attribute>
                <xsl:attribute name="src"> ❶
                    http://static.flickr.com/<xsl:value-of
                    ➥ select="@server"/>/<xsl:value-of select="@id"/>
                    ➥ _<xsl:value-of select="@secret"/>_s.jpg
                </xsl:attribute> ❷
                <xsl:attribute name="alt">
                    <xsl:value-of select="@title"/>
                </xsl:attribute>
                <xsl:attribute name="title">
                    <xsl:value-of select="@title"/>
                </xsl:attribute>
            </xsl:element>
        </td>
    </xsl:template>
    <xsl:template match="err">
        <p class="error">
            Erreur n&deg; <xsl:value-of select="@code"/> :
            <xsl:value-of select="@msg"/>
        </p>
    </xsl:template>
</xsl:stylesheet>
```

Rien d'extraordinaire : on retrouve notamment, de ❶ à ❷, notre code de composition d'URL statique pour le fichier image. Ici, nous utilisons la taille _s, pour la vignette carrée.

Commençons, comme d'habitude, par nous faire une fonction.

Listing 8-51 Notre fonction getPhotosetPhotos

```
function getPhotosetPhotos(id) {
    var url = FLICKR_TEMPLATE.evaluate({
        method: 'photosets.getPhotos',
        extraArgs: '&photoset_id=' + id
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var html = xsltProcess(data, gXSLT.photos);
            new Insertion.Bottom('results', html);
        }
    });
} // getPhotosetPhotos
```

La clé ici, c'est qu'on ne remplace pas le contenu de `results` : on ajoute un nouveau contenu en dessous de l'existant. Notre table `photoset-photos` apparaît donc sous `photoset`.

Il nous reste à appeler cette nouvelle fonction après avoir constitué l'en-tête. Pour laisser à celui-ci le temps de s'afficher d'abord, nous décalerons l'appel de 100 ms.

Listing 8-52 Modification de getPhotoset pour charger les photos

```
function getPhotoset(id) {
    …
        onSuccess: function(requester) {
            …
            $('results').update(html);
            getOwnerInfo(ownerNSID);
            window.setTimeout('getPhotosetPhotos(' + id + ')', 100);
        }
    });
} // getPhotoset
```

Sans les styles et sur une seule ligne, ce n'est pas terrible, mais on sent qu'on est sur la bonne voie (figure 8-15).



Figure 8–15 La table sans les styles, en une seule ligne

Commençons par répartir cela en lignes de 10 vignettes. Pour cela, utilisons un algorithme tout simple :

1. Partons d'un `<tr>…</tr>` contenant toute une série de blocs `<td>…</td>`.
2. Encadrons chaque série de 10 blocs `<td>…</td>` par `<tr>` et `</tr>`.
3. Nous allons fatalement nous retrouver avec, au début, `<tr><tr>` : nous retirerons le doublon.
4. Sur la fin, suivant que nous avions un multiple de 10 vignettes ou non, nous trouverons soit `</tr></tr>`, soit `</tr><td>`. Dans le premier cas, nous retirerons le doublon. Dans le deuxième, nous insérerons un `<tr>` pour démarrer la dernière ligne.
5. C'est tout !

Bien sûr, dans l'intérêt de la modularité et de la lisibilité, nous allons placer ce code dans une fonction, mais cela va être court.

Listing 8-53 Notre fonction adjustTable

```
function adjustTable(html) {
  html = html.replace(/(<td>(.|\n|\r)*?<\/td>){10}/img, function(s) { ❶
    return '<tr>' + s + '</tr>';
  }); ❷
  html = html.replace('<tr><tr>', '<tr>')
    .replace('</tr><td>', '</tr><tr><td>')
    .replace('</tr></tr>', '</tr>');
  return html;
} // adjustTable
```

L'expression rationnelle en ❶ est un peu compliquée. Le groupe entre parenthèses représente une cellule : de `<td>` jusqu'au premier `</td>` rencontré par la suite, sachant qu'entre les deux on peut trouver n'importe quoi, y compris le saut de ligne ou le retour chariot. Le quantificateur `{10}` donne le nombre d'occurrences de ce groupe que nous souhaitons. Enfin, les drapeaux `i`, `m` et `g` (rien à voir avec la balise `img` de HTML !) indiquent respectivement que nous sommes insensibles à la casse, que nous travaillons sur de multiples lignes, et que nous souhaitons ici un remplacement global (de toutes les occurrences de l'expression, pas juste la première).

Nous utilisons en ❷ une fonction plutôt que la syntaxe `$&` de `String.replace` (méthode native de JavaScript). En effet, Safari ne la comprend pas.

Il nous suffit maintenant d'ajuster la ligne de `getPhotosetPhotos` qui insérait le contenu :

```
new Insertion.Bottom('results', adjustTable(html));
```

En rafraîchissant, nous voyons se rapprocher la solution (figure 8-16).



Figure 8–16 Notre tableau à lignes multiples

Ajoutons à présent quelques règles CSS.

Listing 8-54 Ajouts à client.css pour notre grille de vignettes

```
#photoset-photos {
    clear: both;
    border-width: 0;
    border-collapse: collapse;
}
```

```
#photoset-photos tr {
    margin: 0;
    padding: 0;
}

#photoset-photos td {
    padding: 0;
}

#photoset-photos img {
    border: 2px solid silver;
    margin: 0 2px;
}

#photoset-photos img:hover {
    border-color: red;
    cursor: pointer;
}
```

L'impact immédiat est dû à `clear: both` dans `#photoset-photos`, qui place la grille sous l'en-tête. Le reste vise à fournir une bordure et surligner la photo sous le curseur (figure 8-17).



Figure 8–17
Notre grille de vignettes terminée !

Afficher une photo et ses informations

Il ne nous reste plus qu'à fournir un mécanisme pour afficher une photo particulière lorsqu'on clique dessus, en taille par défaut, avec son titre, le nombre de ses commentaires si elle en a, et la liste de ses étiquettes (*tags*).

Nous avons déjà le conteneur pour cet affichage : présent depuis la première heure, notre conteneur `photo` n'attend que ça.

Côté événementiel, il serait stupide, voire suicidaire sur de gros volumes, de définir une inscription par vignette. Nous allons plutôt définir un gestionnaire unique au niveau de `results`. En effet, ce dernier figure toujours dans le DOM ; nous n'avons donc pas besoin de désinscrire ou réinscrire lors de rafraîchissements éventuels.

En revanche, cela impose quelques filtres : nous ne traiterons que les clics effectués sur des éléments `img` qui figurent par ailleurs dans `photoset-photos` (pour éviter notamment le clic sur la photo d'en-tête). Les autres clics seront sans effet.

On peut alors utiliser l'attribut `id` de l'image, que nous avons pris soin de définir dans `photoset_photos.xsl`. Sur la base de cet ID, on peut interroger Flickr avec la méthode `flickr.photos.getInfo` et récupérer les informations que nous souhaitons. Le résultat XML pour une requête de ce type ressemble à ceci.

Listing 8-55 Résultat XML pour une requête flickr.photos.getInfo

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photo id="73723411" secret="a4a9f2246e" server="34"
     ➥ dateuploaded="1134625094" isfavorite="0" license="1"
     ➥ rotation="0" originalformat="jpg">
  <owner nsid="19663157@N00" username="nitot"
       ➥ realname="Tristan Nitot" location="Paris, France" />
  <title>Deb Richardson, aka dria</title>
  <description />
  <visibility ispublic="1" isfriend="0" isfamily="0" />
  <dates posted="1134625094" taken="2005-12-08 21:11:28"
       ➥ takengranularity="0" lastupdate="1155766857" />
  <editability cancomment="0" canaddmeta="0" />
  <comments>2</comments>
  <notes />
  <tags>
  <tag id="627752-73723411-2889" author="19663157@N00"
     ➥ raw="Firefox">firefox</tag>
  <tag id="627752-73723411-1860001" author="19663157@N00"
     ➥ raw="firefox2005offsite">firefox2005offsite</tag>
  </tags>
  <urls>
```

```
    <url type="photopage">http://www.flickr.com/photos/nitot/73723411/
    </url>
  </urls>
</photo>
</rsp>
```

Vous voyez qu'on a tout le nécessaire, et même plus : l'auteur, qu'on connaît déjà dans notre cas, mais aussi la date de publication (en millisecondes), la date de prise de vue (format plus proche du W3DTF), celle de dernière mise à jour, les textes saisis à l'origine pour les étiquettes, etc.

Notre feuille XSLT va extraire la photo en version normale (taille par défaut), et lister le titre, la description, le nombre de commentaires (s'il y en a), et la liste des étiquettes, séparées par des virgules.

Listing 8-56 Notre feuille xsl/photo.xsl, la dernière de l'exemple

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
                ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/photo"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="photo">
       <h2><xsl:value-of select="title"/></h2>
        <xsl:element name="img">
            <xsl:attribute name="src">
                http://static.flickr.com/<xsl:value-of
                ➥ select="@server"/>/<xsl:value-of select="@id"/>
                ➥ _<xsl:value-of select="@secret"/>.jpg
            </xsl:attribute>
        </xsl:element>
        <p class="description"><xsl:value-of select="description"/></p>
        <xsl:if test="comments != 0">
            <p class="commentCount">
                <xsl:value-of select="comments"/>
                commentaire<xsl:if test="comments &gt; 1">s</xsl:if>
            </p>
        </xsl:if>
        <xsl:if test="tags/tag">
            <p class="tags">
                &#201;tiquettes :
                <xsl:for-each select="tags/tag">
                    <xsl:value-of select="."/>
                    <xsl:if test="position() != last()">, </xsl:if>
                </xsl:for-each>
            </p>
```

```
        </xsl:if>
    </xsl:template>
    <xsl:template match="err">
        <p class="error">
            Erreur n&deg; <xsl:value-of select="@code"/> :
            <xsl:value-of select="@msg"/>
        </p>
    </xsl:template>
</xsl:stylesheet>
```

Il n'y a pas de nouveautés là non plus : nous avons déjà géré des pluriels, ou séparé des éléments par des virgules (auteurs et contributeurs dans l'exemple Amazon.fr).

Il nous reste bien sûr à créer la fonction qui va chercher cela. Il s'agit d'un gestionnaire d'événement pour le clic.

Listing 8-57 Notre fonction handleThumbnailClick

```
function handleThumbClick(e) {
    $('photo').hide();
    Event.stop(e);
    var elt = Event.element(e); ❶
    if (elt.tagName.toLowerCase() != 'img' ❷
        || !Element.childOf(elt, 'photoset-photos'))
        return;
    var url = FLICKR_TEMPLATE.evaluate({
        method: 'photos.getInfo',
        extraArgs: '&photo_id=' + elt.id
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var html = xsltProcess(data, gXSLT.photo);
            $('photo-contents').update(html);
            $('photo').show();
        }
    });
} // handleThumbClick
```

La fonction `Event.element` ❶ nous permet de récupérer l'élément réellement soumis à l'événement : comme nous allons inscrire ce gestionnaire auprès de `results`, il pourrait s'agir de n'importe quel élément dans `results`. Les lignes ❷ et suivantes appliquent l'algorithme de filtre que nous avons détaillé plus haut.

Inscrivons maintenant ce gestionnaire en complétant `initPage`.

Listing 8-58 Le gestionnaire est inscrit par initPage

```
function initPage() {
    gXSLT.load(function() {
        getPhotoset(PHOTOSET_ID);
    });
    Event.observe('results', 'click', handleThumbClick);
    new Draggable('photo', {});
} // initPage
```

Pour que cela ait un quelconque intérêt, il faut tout de même jouer un peu avec CSS.

Listing 8-59 Ajouts à client.css pour notre visualiseur de photo

```
#photo {
    left: 2em;
    top: 2em;
    background: white;
    border: 2px solid gray;
}

#photo-closer {
    position: absolute;
    top: 0;
    right: 2px;
    font-weight: bold;
    font-size: small;
    color: maroon;
    cursor: default;
}

#photo-contents {
    margin: 1em 0.5em 0.5em 0.5em;
}

#photo h2 {
    margin: 0 0 1em 0;
}

#photo p {
    margin: 0.5em 0 0 0;
}

#photo .commentCount {
    color: #444;
}
```

```
#photo .tags {
    font-size: small;
    color: gray;
}
```

La figure 8-18 montre le résultat d'un clic.



Figure 8–18
Notre visualisation de photo individuelle

Ooooh !... (et au passage, je vous présente Brendan Eich, l'inventeur de JavaScript). En plus, on peut glisser-déplacer ! Si, si, rappelez-vous le `new Draggable` dans `initPage`.

Si vous cliquez ailleurs que sur le visualiseur, mais toujours dans le rectangle de `results`, le visualiseur est masqué (c'est pourquoi nous avons filtré uniquement *après* avoir fait un `$('photo').hide()`). Ceci dit, ce serait bien que l'on puisse cliquer sur notre petite croix pour fermer, car c'est plus conforme aux attentes de l'utilisateur. Ajoutons donc ceci dans le script.

Listing 8-60 De quoi activer la croix de fermeture

```
function initPage() {
    …
    Event.observe('results', 'click', handleThumbClick);
    Event.observe('photo-closer', 'click', closePhoto);
    new Draggable('photo', {});
} // initPage

function closePhoto(e) {
    Event.stop(e);
    $('photo').hide();
} // closePhoto
```

On rafraîchit, et voilà !

Pour aller plus loin…

Nous ne vous recommanderons pas d'ouvrage particulier, mais vous trouverez ci-après quelques adresses précieuses :

- Les spécifications XPath et XSLT du W3C :
  - http://www.w3.org/TR/xpath
  - http://www.w3.org/TR/xslt
- Pour démarrer, les didacticiels de W3 Schools sont décents :
  - http://www.w3schools.com/xpath/
  - http://www.w3schools.com/xsl/
- Les portails des API vues dans ce chapitre :
  - http://aws-portal.amazon.com/
  - http://www.weather.com/services/xmloap.html
  - http://www.flickr.com/services/api/
- Quelques autres API REST de premier plan :
  - Google : http://code.google.com/apis.html
  - Yahoo! : http://developer.yahoo.com/
  - eBay : http://developer.ebay.com/developercenter/rest

9

L'information à la carte : flux RSS et Atom

Le Web 2.0, c'est aussi la mutualisation des contenus, l'échange standardisé d'informations entre sites, au travers de formats de flux. Les deux principaux formats à ce jour sont RSS et Atom. Les deux sont très employés, aussi analyserons-nous les deux, à titre d'exemple, au travers d'un service de brèves et d'un blog technique très populaire.

Techniquement, ce chapitre ne diffère guère du précédent : vous retrouverez nos requêtes Ajax en GET récupérant un contenu XML, et des transformations XSLT côté client. Vous retrouverez aussi, nécessairement, notre couche serveur dans le rôle de l'intermédiaire permettant d'éviter les problématiques de sécurité sur le navigateur. Cependant, l'utilisation que nous en ferons est différente. Nous ne dialoguerons pas avec une API, mais nous récupérerons un document régulièrement mis à jour.

Aperçu des formats

Voyons d'abord rapidement d'où viennent les formats RSS et Atom, avant de lister les informations qu'ils fournissent généralement. Nous terminerons cette introduction avec un point sur la délicate question de l'incorporation de contenu HTML dans le flux.

Une histoire mouvementée

RSS et Atom ont deux histoires très différentes, quoique liées.

RSS 0.9x et 2.0 : les « bébés » de Dave Winer

RSS est le premier format populaire de syndication. Mis au point par Dave Winer, sur la base de travaux originaux chez Netscape, pour des services comme Radio Userland, le format *Really Simple Syndication* a connu plusieurs versions successives : 0.91, 0.92, 0.93, 0.94 et finalement 2.0 (nous reviendrons dans un instant sur la mystérieuse absence de 1.0 dans cette liste). Tous ces formats partagent les mêmes caractéristiques :

- Ils sont « spécifiés » au moyen d'un vague document en ligne, illustré par quelques exemples, le tout étant très insuffisant pour permettre des implémentations fiables et interopérables.
- Ils sont le travail exclusif de Dave Winer, sans concertation aucune avec la communauté qui grandissait autour de ces formats.
- Enfin, ils sont entièrement dédiés à une utilisation de type blog, ce qui entraîne de sévères limitations pour une utilisation plus large. RSS 2.0 dispose toutefois d'un mécanisme de modules, qui permet l'extension du noyau d'éléments pour des utilisations spécialisées.

La page officielle de RSS 2.0 est http://blogs.law.harvard.edu/tech/rss.

RSS 1.0 : une approche radicalement différente

La version 1.0 est radicalement différente et élaborée par par un groupe de travail officieux n'incluant pas Dave Winer. Ici, RSS signifie *RDF Site Summary*. Les différences sont nombreuses :

- RSS 1.0 est basé sur RDF (*Resource Description Framework*), un langage à balises au cœur des travaux du Web sémantique, qui est assez verbeux et quelque peu redondant, mais offre une structure uniforme pour tous types de données et de relations entre les données.

- RSS 1.0 est conçu pour être extensible, avec un noyau précisément défini et le reste des fonctionnalités réparti dans des modules. Trois modules sont « officiels » : le très vaste module Dublin Core (métadonnées), le module Syndication (fréquence des mises à jour) et le module Content (gestion de contenus incorporés quelconques). On compte plus d'une vingtaine de modules proposés.
- RSS 1.0 est clairement spécifié et ne laisse pas de parts d'ombre.

C'est un bon format, mais il est un peu trop lourd pour son utilisation concrète et engendre trop de complexité pour les cas usuels. En revanche, il est fiable, au sens où il permet de systématiquement lever l'ambiguïté dans ses contenus.

La page officielle de RSS 1.0 est http://web.resource.org/rss/1.0/.

Atom, le fruit de la maturité

Après l'apparition de RSS 1.0, Dave Winer, furieux, a immédiatement renommé sa version 0.94 en RSS 2.0, pour « conserver l'avantage », et a déclaré RSS comme « gelé » : le format était définitif, et toute extension devrait se faire au travers de modules ; par ailleurs, on ne pouvait pas utiliser le terme « RSS » pour désigner un autre format à l'avenir. En somme, il a fait l'enfant.

Le problème, c'est que RSS était très populaire et pourtant très problématique. De très nombreux cas de figure n'étaient pas encodables de façon satisfaisante dans le format : on aboutissait toujours au mieux à du contenu ambigu, au pire à des impasses.

Un groupe de travail s'est donc créé, qui a rapidement trouvé un statut officiel au sein de l'IETF, l'organisme responsable de la plupart des standards et formats qui font vivre Internet. L'objectif était de fournir un format qui soit :

- simple ;
- clair ;
- totalement spécifié (aucune zone d'ombre) ;
- sans ambiguïté aucune dans les contenus produits ;
- au moins équivalent à RSS 2.0 et RSS 1.0 en fonctionnalités ;
- capable d'incorporer facilement n'importe quel contenu, binaire, textuel balisé ou textuel quelconque ;
- standard officiel.

En décembre 2005, sous la forme de la très officielle RFC 4287, ce format a vu le jour, avec en bonus un protocole REST de publication de contenu baptisé *Atom Publishing Protocol*.

La page officielle du format est http://tools.ietf.org/html/rfc4287.

Informations génériques

Tous ces formats de flux fournissent toujours les mêmes informations, peu ou prou, avec plus ou moins de détails, de fiabilité ou d'interopérabilité. On trouve d'abord deux grands niveaux :

- le flux lui-même (le *channel* RSS ou le *feed* Atom) ;
- les entrées individuelles du flux (les *items* RSS ou les *entries* Atom).

D'un flux à l'autre, les entrées individuelles peuvent contenir la totalité de l'information, ou simplement un abrégé, voire une simple référence à la ressource complète sur le Web.

Pour le flux lui-même, on dispose généralement des informations suivantes :

- son titre ;
- une URL de ressource correspondante sur le Web ;
- sa description sommaire ;
- sa langue ;
- ses dates de première parution et de dernière mise à jour ;
- ses informations légales (auteur, copyright, etc.) ;
- une identité visuelle éventuelle (par exemple le logo de l'éditeur).

Pour une entrée individuelle, on trouve souvent :

- son titre ;
- une URL vers la ressource complète ;
- un identifiant unique ;
- tout ou partie de son contenu ;
- ses dates de première parution et de dernière mise à jour ;
- des informations de catégorie ;
- son ou ses auteur(s).

Le casse-tête du contenu HTML

Tous ces formats de flux reposent au final sur XML. Il s'agit de documents balisés, qui se doivent de constituer des document XML correctement formés. Ils répondent à une grammaire, théoriquement formalisée dans un document dédié (DTD, schéma XML ou Relax NG) ; toutefois, RSS 2.0 n'a pas de grammaire formalisée.

Les contenus HTML et XHTML utilisent eux aussi des balises. En HTML, le balisage ne constitue pas forcément du XML correctement formé : on peut se passer de fermer des balises, d'encadrer des valeurs d'attributs par des guillemets, d'utiliser

une casse spécifique… On peut même entrelacer les éléments plutôt que de respecter une imbrication cohérente ! Un tel contenu, utilisé tel quel, « casserait » donc le flux.

Quand bien même le contenu est du XHTML valide, sa grammaire ne fait pas partie de celle du format de flux dans lequel on souhaite l’incorporer. Pour permettre aux analyseurs XML de s’y retrouver en lisant le flux, il faut avoir recours soit à l’encodage (typiquement, RSS 2.0), soit aux espaces de noms (solution 100 % XML employée par RSS 1.0 et Atom), ce qui exige toutefois un traitement plus perfectionné côté client.

Au final, RSS 2.0 trouve ici ses limites, avec un problème dit du « double encodage ». Sans entrer dans les détails, on peut tomber sur des cas où, en analysant un contenu d’élément `description`, on n’est pas en mesure de déterminer avec certitude si un fragment doit être « décodé » ou non.

Atom et RSS 1.0 n’ont pas ce problème. Atom en particulier, au travers de son élément `atom:content`, permet l’encapsulation facile de n’importe quel type de contenu : textuel, XHTML, et même binaire.

Passons maintenant à la pratique.

Récupérer et afficher un flux RSS 2.0

Pour illustrer l’utilisation d’un flux RSS 2.0 (qui est de loin la variante RSS la plus employée), nous allons consulter les brèves « Client Web » publiées par le Journal du Net Développeurs, publication en ligne du Benchmark Group.

Ce flux contient des informations succinctes (ce qu’on appelle classiquement des « brèves ») centrées sur l’univers des technologies web côté client (typiquement le sujet de ce livre). L’adresse du flux RSS 2.0 pour ces brèves est http://developpeur.journaldunet.com/rss/breve/client-web/. Si vous vous y rendez, vous tombez pourtant sur un document XHTML tout ce qu’il y a de plus banal. Où donc se trouve le flux ?

Devant vous. Vous êtes en fait en train de consulter un document XML doté d’une feuille XSLT appliquée directement par votre navigateur (en tout cas sur Firefox, Opera et MSIE ; Konqueror par exemple n’effectue pas la transformation).

Nous utiliserons quant à nous notre propre feuille XSLT et reformaterons d’ailleurs les informations de dates et les titres.

Format du flux

En examinant le flux RSS en question, vous découvrez à peu près ceci (nous avons réindenté, abrégé des parties et retiré des fragments, dans un objectif global de lisibilité de l'extrait).

Listing 9-1 Fragments du flux RSS du JDN Développeurs

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet type="text/xsl"
 ➥ href="http://developpeur.journaldunet.com/rss/html_include/style/rss.xsl"?>
<rss version="2.0"> ❶
  <channel> ❷
    <title>Les br&#232;ves du Journal du Net…</title> ❸
    <link>http://developpeur.journaldunet.com/</link>
    <description>Les br&#232;ves du Journal du…</description>
    <language>fr</language>
    <pubDate>Mon, 04 Sep 2006 17:23:09 +0200</pubDate>
    <lastBuildDate>Fri, 08 Sep 2006 19:00:26 +0200</lastBuildDate> ❹
    <copyright>&#xA9; Benchmark Group</copyright>
    <image> ❺
      <title>Les br&#232;ves du Journal du Net…</title>
      <link>http://developpeur.journaldunet.com/</link>
      <description>Les br&#232;ves du Journal du…</description>
      <url>http://developpeur.j…/logo_jdn_developpeurs.gif</url>
    </image>
    <item> ❻
    <title>Client Web &#62; Firefox 2 en beta 2</title>
    <link>http://…ent-web/4614/firefox-2-en-beta-2.shtml</link>
      <guid>http://…ent-web/4614/firefox-2-en-beta-2.shtml</guid>
    <description>La prochaine &#233;volution du navigateur Open Source
se dessine &#224; l'horizon : outre un rafra&#238;chissement de son
interface graphiq...</description> ❼
      <pubDate>Mon, 04 Sep 2006 17:23:09 +0200</pubDate> ❹
      <category domain="http://journaldunet.com/breve/client-web/">
        ➥ Client Web</category>
    </item>
    …
  </channel>
</rss>
```

L'élément racine est `rss` ❶, originellement pour laisser la porte ouverte à un emploi multicanaux, même si cela n'a jamais eu lieu. On descend donc encore d'un niveau dans `channel` ❷, qui représente le flux à proprement parler.

Les éléments `title`, `link` et `description` ❸ sont les trois éléments incontournables de RSS. Pour un `item`, `description` fournit soit une version abrégée du contenu, soit le contenu total. Ici, le JDN Développeurs a opté pour un contenu raccourci, sans

HTML ❼, ce qui évite bien des problèmes. Notez toutefois que les accents sont encodés en dépit de la déclaration du jeu de caractères dans le prologue XML (première ligne), afin de se « blinder » contre les couches serveur ou client mal faites. On utilise ici les codes Unicode des caractères accentués (233 pour le « é », 224 pour le « à », 238 pour le « î », etc.).

En RSS, les dates et heures ❹ sont fournies dans le bon vieux format de la RFC 822 (pas la 2822, qui lui a succédé en avril 2001 et étend les possibilités ; la 822, ce dinosaure datant... du 13 août 1982, dix ans avant le Web !). Cet ancien format a toutefois un avantage : il est facile à interpréter en JavaScript avec l'objet `Date`.

Après les métadonnées du flux lui-même, on trouve son identité visuelle, exprimée par un logo du JDN Développeurs ❺ et enfin un élément `item` ❻ par entrée individuelle ; ici donc, un `item` par brève.

Préparer le terrain

Commencez par faire un répertoire de travail `rss2`, dans lequel on retrouve notre bon vieux `serveur.rb` (oui, le même que depuis le début du chapitre 8), ainsi bien sûr que `docroot` et ses sous-répertoires `ajaxslt` et `xsl`. Nous nommerons la feuille XSLT `breves.xsl`. Réduisez-la à un squelette valide.

Dans `docroot`, nous retrouvons bien sûr `prototype.js`, `client.js`, `client.css`, `spinner.gif` et `index.html`.

La page est absolument triviale : elle contient seulement l'indicateur de chargement et de mise en forme ainsi qu'un conteneur de résultats. Même pas de titre ! C'est logique : nous utiliserons dynamiquement le titre du flux. Voici `index.html`.

Listing 9-2 Notre page index.html, toute simple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Les brèves Client Web du JDN Développeurs</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="ajaxslt/misc.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt/xpath.js"></script>
    <script type="text/javascript" src="ajaxslt/xslt.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
```

```
<body>

<div id="indicator" style="display: none;"></div>

<div id="results"></div>

</body>
</html>
```

Voyons à présent le début de notre feuille de styles. Nous y ajouterons le nécessaire après avoir obtenu le contenu XHTML que nous souhaitons.

Listing 9-3 Premier jet de client.css

```
body {
    font-family: sans-serif;
    font-size: 12pt;
}

#indicator {
    position: absolute;
    top: 10px; left: 10px;
    height: 16px; width: 20em;
    color: gray;
    font-size: 14px;
    line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
}

#results {
    margin-top: 34px;
}
```

Pour finir, voyons le squelette classique de `client.js`.

Listing 9-4 Squelette classique restant pour client.js

```
var xsltSheet;

function showIndicator() {
    with ($('indicator')) {
        update('');
        show();
    }
} // showIndicator
```

```
function hideIndicator() {
    $('indicator').hide();
} // hideIndicator

function initPage() {
    // Prochainement, le chargement XSLT puis flux...
} // initPage

Ajax.Responders.register({ onException: function(requester, e) {
    $('indicator').hide();
    alert(e);
}});

logging__ = false;

Event.observe(window, 'load', initPage);
```

Tout ceci doit vous sembler désormais très familier. Passons maintenant à la définition de notre transformation.

La feuille XSLT

La feuille n'est pas très compliquée. Nous allons produire un en-tête avec le logo, le titre et la date de dernière mise à jour, puis une liste des brèves, avec leur titre en lien et la description courte. Dans notre cas, une liste de définitions (éléments `dl`, `dt` et `dd`) est sémantiquement très appropriée.

Voici la feuille. Pour faciliter la lecture, nous avons découpé le tout en trois *templates*, qui correspondent à l'en-tête, aux brèves et au pied de liste (informations de copyright).

Listing 9-5 Notre feuille xsl/breves.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
        ➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <xsl:apply-templates select="/rss/channel"/>
</xsl:template>
<xsl:template match="channel">
  <xsl:call-template name="header"/>
  <xsl:call-template name="body"/>
  <xsl:call-template name="footer"/>
</xsl:template>
```

```
<xsl:template name="header">
  <div id="header">
    <p>
      <xsl:element name="img">
        <xsl:attribute name="id">logo</xsl:attribute>
        <xsl:attribute name="src"><xsl:value-of
               ➥ select="image/url"/></xsl:attribute>
        <xsl:attribute name="alt"><xsl:value-of
               ➥ select="image/description"/></xsl:attribute>
      </xsl:element>
    </p>
  <h1><xsl:value-of select="title"/></h1>
    <p id="lastBuildDate">
      Derni&egrave;re mise &agrave; jour :
      <span class="timestamp"><xsl:value-of
               ➥ select="lastBuildDate"/></span>
    </p>
  </div>
</xsl:template>
<xsl:template name="body">
  <dl>
    <xsl:for-each select="item">
      <dt>
        <xsl:element name="a">
          <xsl:attribute name="href"><xsl:value-of select="link"/>
          </xsl:attribute>
          <xsl:value-of select="title"/>
        </xsl:element>
        <span class="pubDate">
          &middot;
          <span class="timestamp"><xsl:value-of select="pubDate"/>
          </span>
        </span>
      </dt>
      <dd>
        <p><xsl:value-of select="description"/></p>
      </dd>
    </xsl:for-each>
  </dl>
</xsl:template>
<xsl:template name="footer">
  <p class="footer"><xsl:value-of select="copyright"/></p>
</xsl:template>
</xsl:stylesheet>
```

Nous avons souligné les éléments importants avec leurs classes et ID, pour vous permettre de bien voir les relations avec les règles CSS que nous ajouterons par la suite.

Chargement et formatage du flux

Il est temps d'ajouter le désormais traditionnel code de chargement de feuille XSLT puis de flux. N'ayant qu'une seule feuille, il est inutile de recourir à des objets dédiés comme le `gXSLT` de l'exemple Flickr au chapitre 8. Voici notre `initPage`.

Listing 9-6 Notre fonction initPage terminée

```
function initPage() {
    new Ajax.Request('xsl/breves.xsl', {
        method: 'get',
        indicator: 'indicator',
        onSuccess: function(requester) {
            xsltSheet = xmlParse(requester.responseText);
            getFeed();
        }
    });
} // initPage
```

Par souci de clarté et de modularité, nous avons délégué le chargement et le traitement du flux à une fonction à part, `getFeed`. La voici.

Listing 9-7 Notre fonction getFeed

```
FEED_URL = 'http://developpeur.journaldunet.com/rss/breve/client-web/';

function getFeed() {
    $('results').update('');
    showIndicator();
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(FEED_URL),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('indicator').update('Mise en forme&#8230;');
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsltProcess(data, xsltSheet);
                $('results').update(html);
                hideIndicator();
            }, 10);
        }
    });
} // getFeed
```

Il n'y a là que du grand classique. Voilà, nous avons le minimum nécessaire. Lançons `serveur.rb` et naviguons jusqu'à http://localhost:8042 (figure 9-1).



Figure 9–1
Notre flux RSS chargé

Évidemment, un peu de CSS ne fait jamais de mal.

Listing 9-8 Des règles CSS pour rendre tout ceci plus joli

```
#results h1 {
    font-family: Georgia, serif;
    font-size: 140%;
    color: #555;
}

p#lastBuildDate {
    font-size: smaller;
    color: gray;
}

#results p.footer {
    font-size: small;
    color: gray;
    border-top: 1px solid silver;
    margin: 1em 0;
}

#results dl {
    margin: 1em 0 0 1em
}
```

```
#results dt a {
    font-weight: bold;
    color: green;
}

#results dt a:visited {
    color: #050;
}

#results span.pubDate {
    color: gray;
    font-size: smaller;
    font-weight: normal;
}

#results dd {
    margin-left: 1em;
}

#results dd p {
    margin: 0.3em 0 1em 0;
    font-size: 90%;
    color: #444;
}
```

Rafraîchissons, nous obtenons la figure 9-2.



Figure 9–2
C'est déjà beaucoup mieux.

Il nous reste tout de même à nous occuper des dates (le format actuel n'est pas très agréable pour des francophones) et des titres (ce préfixe « Client Web > » est agaçant).

Ajustements des dates et titres

Comme d'habitude, nous allons dédier une fonction `adjustData` à ces traitements et utiliser des expressions rationnelles. Si vous examinez notre feuille XSLT, vous voyez que nous avons isolé toute date produite dans un `span` de classe `timestamp`. Quant aux titres, ils sont dans des éléments `a` dont le libellé commence par « `Client Web >` ». Le code de traitement donne ceci.

Listing 9-9 Notre ajustement de contenus

```
RE_TITLE = '(<a .*?>)Client Web\\s*&gt;\\s*(.*?)(</a>)';
RE_TIMESTAMP = '(<span class="timestamp">)(.*?)(</span>)';

function adjustData(html) {
    var html = html.gsub(RE_TIMESTAMP, function(match) {
        match[2] = new Date(match[2]).toLocaleString();
        return match[1] + match[2] + match[3];
    });
    html = html.replace(new RegExp(RE_TITLE, 'img'), '$1$2$3');
    return html;
} // adjustData
```



Figure 9–3
Notre affichage de flux RSS prêt à l'emploi

Évidemment, il ne faut pas oublier d'ajuster `getFeed`.

Listing 9-10 Le fragment de getFeed utilisant l'ajustement

```
var tmr = window.setTimeout(function() {
    …
    $('results').update(adjustData(html));
    hideIndicator();
}, 10);
```

Après un petit rafraîchissement, nous obtenons la figure 9-3.

Affichage plus avancé et flux Atom 1.0

À présent que nous avons vu un exemple simple autour de RSS 2.0, il est temps de voir un exemple plus complet, autour d'Atom. Nous utiliserons l'incontournable Standblog, le blog de Tristan Nitot, président de Mozilla Europe et fervent évangéliste des standards du Web depuis bien des années.

Le blog de Tristan tourne sur Dotclear (http://www.dotclear.com), l'excellent moteur de blog conforme aux standards d'Olivier Meunier. Comme la plupart des outils de blog, Dotclear fournit d'office des flux Atom et permet au blogueur de choisir s'il (ou elle) souhaite incorporer le texte complet des billets dans le flux, ou simplement un abrégé.

Tristan choisit le texte intégral, balisage XHTML compris, ce qui sert parfaitement notre exemple : cela ouvre des possibilités amusantes côté client, avec des effets script.aculo.us, pour afficher d'abord les textes abrégés, et faire apparaître sur demande, de façon un peu vivante, les contenus complets.

En revanche, cela va nous demander un peu de travail supplémentaire. En effet, Dotclear génère des flux Atom avec des contenus en mode HTML encodé (un des types possibles pour l'élément `atom:content`), ce qui signifie que le HTML est encodé dans un nœud texte, de façon assez similaire à RSS (mais sans l'ambiguïté, le format précisant qu'on n'utilise pas de double encodage).

Ce nœud ne peut pas être transformé en HTML par XSLT : c'est à notre JavaScript de manipuler le texte du fragment XHTML résultat de la transformation. Et cette transformation pose quelques problèmes techniques de portabilité, qui donneront parfois du code un peu « à la main ». Si Dotclear produisait des éléments `atom:content` recourant aux espaces de noms, nous n'aurions rien eu à faire. Olivier a certainement une bonne raison d'avoir procédé ainsi.

Si ce sujet (atom:content et ses différents modes) vous intéresse, il est traité dans tous les détails, avec des exemples, pages 133 à 142 de l'ouvrage sur RSS et Atom mentionné en fin de chapitre.

Notre flux Atom

Commençons par examiner le flux Atom du Standblog. Voici une vue fragmentée, ré-indentée, etc..

Listing 9-11 Vue filtrée du flux Atom du Standblog

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<feed xmlns="http://www.w3.org/2005/Atom" ❶
      xmlns:sy="http://purl.org/rss/1.0/modules/syndication/" ❷
      xml:lang="fr">
  <title>Standblog</title> ❸
  <link rel="alternate" type="text/html"
       ➥ href="http://standblog.org/blog/"/>
  <link rel="self" href="http://standblog.org/dotclear/atom.php"/>
  <id>tag:standblog.org,2006:/blog/</id>
  <updated>2006-09-08T20:16:11+02:00</updated> ❹
  <generator version="1.2.5" uri="http://www.dotclear.net/">DotClear
  </generator>
  <sy:updatePeriod>daily</sy:updatePeriod> ❷
  <sy:updateFrequency>1</sy:updateFrequency>
  <sy:updateBase>2006-09-08T20:16:11+02:00</sy:updateBase>
  <entry xml:lang="fr"> ❶
    <title>En vrac, vite fait...</title>
    <link rel="alternate" type="text/html" href="http://standblog.org/
                      ➥blog/2006/09/08/93114894-en-vrac-vite-fait" />
    <updated>2006-09-08T20:16:11+02:00</updated>
    <id>tag:standblog.org,2006-09-08:/dotclear/93114894</id>
    <author><name>Tristan</name></author>
    <category term="En-vrac" label="En vrac"/>
    <summary>...oui, encore plus vite fait que d'habitude :-)
Michel de Guilhermier aime iGraal (merci Yann pour le lien !) ;
Voici une offre d'emploi pour un développeur XUL ;
The difficulty of simplicity... Je n'aurais pas dis mieux !
Adobe laisse tomber son plug-in SVG... Certes, Opera et...
    </summary>
    <content type="html"> &lt;p&gt;...oui, encore plus vite fait que
d'habitude :-)&lt;/p&gt; ❺
&lt;ul&gt;
&lt;li&gt;&lt;a href=&quot;http://micheldeguilhermier.typepad.com/
mdegblog/2006/09/brand_new_conna.html&quot;
hreflang=&quot;fr&quot;&gt;Michel de Guilhermier aime iGraal&lt;/a&gt;
(merci Yann pour le lien !)&amp;nbsp;;&lt;/li&gt;
```

```
&lt;li&gt;Voici une &lt;a href=&quot;http://fredericdevillamil.com/
articles/2006/09/04/offre-demploi-d%C3%A9veloppeur-php-xul&quot;
hreflang=&quot;fr&quot;&gt;offre d'emploi pour un développeur XUL&lt;/
a&gt;&amp;nbsp;;&lt;/li&gt;

&lt;li&gt;&lt;a href=&quot;http://www.allpeers.com/blog/2006/09/07/the-
difficulty-of-simplicity/&quot; hreflang=&quot;en&quot;&gt;The
difficulty of simplicity&lt;/a&gt;... Je n'aurais pas dis
mieux&amp;nbsp;!&lt;/li&gt;
&lt;li&gt;&lt;a href=&quot;http://georezo.net/forum/
viewtopic.php?pid=56305#p56305&quot; hreflang=&quot;fr&quot;&gt;Adobe
laisse tomber son plug-in SVG&lt;/a&gt;... Certes, Opera et Firefox
intègrent une partie de la spécification, mais ça n'est pas une bonne
raison à mon sens. Il faut dire que le SVG est en concurrence partielle
avec Flash, lequel dépend d'Adobe depuis le rachat de Macromedia... la
&lt;a href=&quot;http://www.adobe.com/svg/pdfs/ASV_EOL_FAQ.pdf&quot;
hreflang=&quot;en&quot;&gt;FAQ de finde vie du plug-in SVG (format
PDF)&lt;/a&gt; est très révélatrice&amp;nbsp;;&lt;/li&gt;

&lt;li&gt;Merci à Filip qui me signale que ma &lt;a href=&quot;&quot;
hreflang=&quot;fr&quot;&gt;photo de la tour Eiffel&lt;/a&gt; est
reprise &lt;a href=&quot;http://www.smartphone.net/smartphonethoughts/
software_detail.asp?id=2520&quot; hreflang=&quot;en&quot;&gt;ici&lt;/
a&gt;. Dans la même série, elle a déjà été publiée en Nouvelle Zélande,
et de nombreuses photos de votre serviteur servent d'exemples dans un
&lt;a href=&quot;http://gimp4you.eu.org/livre/&quot;
hreflang=&quot;fr&quot;&gt;nouveau livre sur Gimp&lt;/a&gt;.&lt;/li&gt;

&lt;/ul&gt;
    </content>
  </entry>
  …
</feed>
```

Vous voyez en ❶ les deux éléments clés du flux : `feed`, élément racine effectif, qui représente le flux (on casse donc d'entrée de jeu la compatibilité avec RSS), et `entry`, qui représente une entrée individuelle.

Les lignes marquées ❷ montrent un module d'extension par espaces de noms en action : le module officiel Syndication. Le préfixe `sy` est associé à l'URL officielle de l'espace de noms pour le module, et les éléments issus de cet espace emploient le préfixe. Il s'agit du mécanisme fondamental d'extension en XML. Le « X » de XML, *eXtensible*, c'est beaucoup grâce à lui.

Vous remarquez en ❸ qu'on retrouve `title` et `link`, même si en Atom, `link` a un rôle beaucoup plus clair, tout en restant polyvalent, que dans RSS. En revanche, plus de schizophrénie avec `description`, tantôt résumé et tantôt contenu complet : Atom clarifie les rôles avec `summary` et `content`.

Vous observez en ❹ que les dates/heures en Atom utilisent un format récent, de plus en plus répandu : le W3DTF (*W3C Date/Time Format*). C'est très bien, mais l'objet JavaScript Date n'a pas encore suivi le mouvement : ne serait-ce qu'en raison de la représentation du décalage GMT, nous ne pourrons nous contenter d'une expression rationnelle pour reformater et interpréter les dates : il faudra les décoder manuellement.

Enfin, remarquez en ❺ l'élément content et son attribut type, qui vaut ici html. Dans ce mode, le contenu est encodé : le client *sait* qu'il doit décoder ce contenu une et une seule fois (c'est-à-dire que si le décodage produit de nouvelles entités, par exemple avec > qui donne >, on en reste là).

Tristan étant parfois prolifique, on se retrouve de temps en temps avec de grosses grappes de HTML encodé à transformer, en JavaScript, en HTML normal. En raison de la taille globale du HTML obtenu par XSLT (qui représente tout le flux), nous aurions quelques problèmes avec les expressions rationnelles sur certains navigateurs ; ainsi, Konqueror 3.5.2 « planterait » purement et simplement, tandis que Firefox se bornerait à 4 Ko de texte pour certaines opérations relatives au décodage des entités HTML... Il s'agit donc de tâches que nous devrons à nouveau effectuer manuellement...

Préparation de notre lecteur de flux

Allez, c'est parti. Vous avez dû tout retenir maintenant, mais nous vous redonnons tout de même les étapes habituelles : faites un répertoire de travail atom, dans lequel vous installez notre vaillant serveur.rb (toujours le même), le répertoire docroot et ses sous-répertoires ajaxslt et xsl. La feuille XSLT sera standblog.xsl. Réduisez-la à un squelette valide.

Dans docroot, vous placez bien sûr prototype.js, client.js, client.css, spinner.gif et index.html, comme toujours.

La page est toujours aussi triviale : on a juste l'indicateur de chargement et de mise en forme, et un conteneur de résultats. Au titre près, c'est exactement celle de l'exemple précédent, avec tout de même un chargement script.aculo.us en plus, en prévision d'effets à venir (par conséquent, ajoutez scriptaculous.js et effects.js dans docroot). Voici le HTML.

Listing 9-12 Notre page index.html, à peine différente de la précédente

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      ➥ xml:lang="fr-FR">
```

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
          ➥ charset=iso-8859-15" />
    <title>Le Standblog</title>
    <link rel="stylesheet" type="text/css" href="client.css"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="scriptaculous.js?load=effects">
    </script>
    <script type="text/javascript" src="ajaxslt/misc.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt/xpath.js"></script>
    <script type="text/javascript" src="ajaxslt/xslt.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<div id="indicator" style="display: none;"></div>

<div id="results"></div>

</body>
</html>
```

Le début de notre feuille de styles est identique à l'exemple précédent. Nous y ajouterons le nécessaire après avoir obtenu le contenu XHTML que nous souhaitons, lequel différera tout de même beaucoup.

Listing 9-13 Premier jet de client.css

```
body {
    font-family: sans-serif;
    font-size: 12pt;
}

#indicator {
    position: absolute;
    top: 10px; left: 10px;
    height: 16px; width: 20em;
    color: gray;
    font-size: 14px;
    line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
}

#results {
    margin-top: 34px;
}
```

Côté script, nous pouvons conserver la base de l'exemple précédent, notamment la constante de l'URL du flux (mais évidemment, ce n'est plus la même URL), et l'expression rationnelle isolant les dates, en plus du code habituel.

Listing 9-14 Squelette classique restant pour client.js

```
FEED_URL = 'http://standblog.org/dotclear/atom.php';

RE_TIMESTAMP = '(<span class="timestamp">)(.*?)(</span>)';

var xsltSheet;

function showIndicator() {
    with ($('indicator')) {
        update('');
        show();
    }
} // showIndicator

function hideIndicator() {
    $('indicator').hide();
} // hideIndicator

function initPage() {
    // Prochainement, le chargement XSLT puis flux...
} // initPage

Ajax.Responders.register({ onException: function(requester, e) {
    $('indicator').hide();
    alert(e);
}});

logging__ = false;

Event.observe(window, 'load', initPage);
```

Passons maintenant à la définition de notre feuille XSLT.

La feuille XSLT et le formatage

Nous avons conservé le principe des trois *templates* d'en-tête, de corps et de pied. Plus de logo ici, mais toujours des entrées individuelles dans une liste de définitions (`dl`, `dt`, `dd`). En revanche, nous avons deux blocs de texte par entrée :

- un paragraphe de résumé visible, avec sur la fin du texte un lien d'affichage du contenu complet ;
- et un conteneur initialement masqué pour ce fameux contenu.

Nous ajoutons aussi à chaque entrée sa langue, fournie dans l'attribut standard `xml:lang` des éléments `entry`. C'est important sur le Standblog, car les billets sont aussi souvent en anglais qu'en français.

Voici la feuille...

Listing 9-15 Notre feuille xsl/standblog.xsl

```
<xsl:template match="/">
  <xsl:apply-templates select="/feed"/>
</xsl:template>
<xsl:template match="feed">
  <xsl:call-template name="header"/>
  <xsl:call-template name="body"/>
  <xsl:call-template name="footer"/>
</xsl:template>
<xsl:template name="header">
  <div id="header">
    <h1><xsl:value-of select="title"/></h1>
    <p id="lastBuildDate">
      Derni&egrave;re mise &agrave; jour :
      <span class="timestamp"><xsl:value-of select="updated"/>
      </span>
    </p>
  </div>
</xsl:template>
<xsl:template name="body">
  <dl>
    <xsl:for-each select="entry">
      <xsl:element name="dt">
        <xsl:copy-of select="@xml:lang"/> ❶
        <xsl:element name="a">
          <xsl:attribute name="href">
              <xsl:value-of select="link[@type='text/html']/@href"/> ❷
          </xsl:attribute>
          <xsl:value-of select="title"/>
        </xsl:element>
        <span class="pubDate">
          &middot;
          <span class="timestamp"><xsl:value-of select="updated"/>
          </span>
        </span>
      </xsl:element>
      <dd>
        <xsl:copy-of select="@xml:lang"/> ❶
        <p class="summary">
          <xsl:value-of select="summary"/>
          <span class="toggler">[ ❸
            <xsl:element name="a">
```

```
              <xsl:attribute name="href">
                <xsl:value-of select="link[@type='text/html']/@href"/>
              </xsl:attribute>
              Voir tout l'article
            </xsl:element>
          ]</span> ❹
        </p>
        <div class="content" style="display: none">
          <div><xsl:value-of select="content"/></div>
        </div>
      </dd>
    </xsl:for-each>
  </dl>
</xsl:template>
<xsl:template name="footer">
  <p class="footer"><xsl:value-of select="copyright"/></p>
</xsl:template>
</xsl:stylesheet>
```

C'est la première fois que nous utilisons `xsl:copy-of` ❶ dans une feuille XSLT. Cette instruction copie un fragment XML dans notre résultat, en tant que XML toujours : ici, nous récupérons l'attribut `xml:lang` tel quel, ce qui n'a de sens que dans un `xsl:element` (et nous y sommes : respectivement `dt` et `dd`). Nous indiquons ainsi la langue des textes de l'entrée, ce qui sera notamment très utile aux logiciels lecteurs d'écran.

La ligne ❷ illustre l'obtention de l'URL du billet sur le Web : en Atom, les éléments `link` fournissent l'URL dans leur attribut `href`, et non ailleurs. Nous prenons bien soin de choisir un lien typé `text/html`, qui représente normalement la ressource (idéalement, il faudrait en plus tester que l'attribut `rel` vaut `alternate`) : il pourrait y en avoir d'autres (feuilles de styles CSS suggérées, relations hiérarchiques dans un ensemble de documents, etc.).

Les lignes ❸ à ❹ créent les liens de bascule d'affichage, en fin de chaque texte abrégé. Pour l'accessibilité, il est bon que ces liens, dont la vocation est d'afficher le texte complet, amènent naturellement (attribut `href`) sur le billet. Nous intercepterons leur déclenchement avec de l'*unobstrusive JavaScript* pour afficher le contenu déjà chargé, mais initialement masqué. C'est le même principe que pour un pop-up accessible.

Charger la feuille et le flux

Notre feuille est prête ; il nous reste à la charger, puis à récupérer le flux et appliquer la feuille.

Voici déjà `initPage`.

Listing 9-16 Notre fonction initPage qui charge la feuille puis le flux

```
function initPage() {
    new Ajax.Request('xsl/standblog.xsl', {
        method: 'get',
        indicator: 'indicator',
        onSuccess: function(requester) {
            xsltSheet = xmlParse(requester.responseText);
            getFeed();
        }
    });
} // initPage
```

Le code est presque identique à celui de l’exemple précédent : seul le nom de la feuille a changé. Voyons à présent `getFeed` et notre éternelle `adjustData`, qui ne fait rien pour l’instant.

Listing 9-17 Le chargement du flux et la transformation avec getFeed

```
function adjustData(html) {
    return html;
} // adjustData

function getFeed() {
    $('results').update('');
    showIndicator();
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(FEED_URL),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('indicator').update('Mise en forme&#8230;');
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsltProcess(data, xsltSheet);
                $('results').update(adjustData(html));
                hideIndicator();
            }, 10);
        }
    });
} // getFeed
```

Et voilà ! Nous pouvons lancer la couche serveur et tester (figure 9-4).



Figure 9–4
Le Standblog, un peu nu, dans notre navigateur

Commençons par habiller notre Standblog en ajoutant quelques règles CSS.

Listing 9-18 Ajouts à client.css pour habiller le flux

```
#results h1 {
    font-family: Georgia, serif;
    font-size: 140%;
    color: #555;
}

p#lastBuildDate {
    font-size: smaller;
    color: gray;
}

#results p.footer {
    font-size: small;
    color: gray;
    border-top: 1px solid silver;
    margin: 1em 0;
}

#results dl {
    margin: 1em 0 0 1em
}
```

```
#results dt a {
    font-weight: bold;
    color: green;
}

#results dt a:visited {
    color: #050;
}

#results span.pubDate {
    color: gray;
    font-size: smaller;
    font-weight: normal;
}

#results dd {
    margin-left: 1em;
    font-size: 90%;
    color: #444;
}

#results dd p {
    margin: 0.3em 0 1em 0;
}

#results dd div.content {
    margin: 0.3em 0 1em 0;
    border: 1px solid gray;
    background: #ddd;
    padding: 1em;
}
```

Le résultat est présenté sur la figure 9-5.

C’est déjà mieux, même si les dates W3DTF détonnent un peu. Occupons-nous maintenant de donner vie aux liens Voir tout l’article, qui pour l’instant sont des liens classiques.

Afficher dynamiquement les billets complets

Une fois le flux chargé, transformé et inséré dans le DOM, nous devons réagir aux clics sur les liens de bascule pour afficher le contenu complet (présent dans un `div` caché après le paragraphe abrégé) et masquer le texte abrégé.

En même temps, utilisons script.aculo.us pour rendre l’ensemble plus joli : un fondu jusqu’à disparition de l’abrégé, tandis que la version complète apparaît en défilant vers le bas, le tout synchronisé. Cela nous permettra de réviser le chapitre 7.



Figure 9–5
L'affichage initial, habillé

Définissons d'abord une fonction chargée de récupérer tous les liens de bascule et de leur affecter un gestionnaire d'événement unique. C'est l'occasion de voir `$$` en action.

Listing 9-19 Fonction bindTogglers

```
function bindTogglers() {
    $$('#results .toggler a').each(function(link) {
        Event.observe(link, 'click', handleToggler);
    });
} // bindTogglers
```

Nous allons donc écrire un gestionnaire `handleToggler`. Notez la puissance de `$$`, qui nous fournit un énumérable de tous les éléments `a` contenus dans un élément de classe `toggler` au sein du conteneur `results`. Imaginez juste à quoi ressemblerait cette fonction *sans* Prototype...

Voyons maintenant notre fonction `handleToggler`, qui met elle aussi Prototype lourdement à contribution, en utilisant notamment des nouveautés de la version 1.5.0_rc1.

Listing 9-20 Le gestionnaire unique handleToggler

```
function handleToggler(e) {
    Event.stop(e);
    var toggler = $(Event.element(e)).up('p');
```

```
    new Effect.Parallel([
        new Effect.BlindDown(toggler.next('div')),
        new Effect.Fade(toggler)
    ], { duration: 2.0 });
} // handleToggler
```

Voilà beaucoup de Prototype en très peu de lignes. Nous commençons par interrompre l'événement pour éviter la navigation normale après clic. Nous récupérons ensuite l'élément concerné (le lien qui a été cliqué), nous le passons à `$` pour obtenir un élément garanti étendu, sur lequel nous pouvons donc appeler `up`, ici pour remonter jusqu'au premier ancêtre de balise `p`.

Ensuite, nous créons une exécution parallèle de deux effets, sur une durée de deux secondes : un défilement bas (contenant, pas contenu) du contenu complet, masqué jusqu'ici (attribut `style="display: none"` dans le HTML), et un fondu jusqu'à disparition du paragraphe abrégé (lien de bascule compris).

Notez comment nous désignons le contenu complet : en partant du paragraphe abrégé, et en suivant ses nœuds frères vers le bas jusqu'à tomber sur un `div`. Remarquez au passage que la fonction `up` renvoyant un élément garanti étendu dans `toggler`, nous pouvons appeler `next` directement sur celui-ci (sans encadrer par `$`, par exemple).

Le HTML fourni par la feuille XSLT ajoute un `div` à l'intérieur de celui manipulé ici, afin de permettre l'effet `SlideDown` au lieu de `BlindDown`, qui est conceptuellement plus sympathique, mais a tendance à beaucoup clignoter sur certaines configurations. `BlindDown` est plus stable. Notez aussi que `Fade`, comme tout effet utilisant `opacity`, ne marche pas dans les versions actuelles de Konqueror : les utilisateurs de ce navigateur verront juste le paragraphe abrégé disparaître d'un coup, en fin d'effet.

Il ne nous reste qu'à appeler `bindTogglers` après insertion du HTML du flux dans le DOM de la page, au sein de `getFeed`.

Listing 9-21 Ajustement de getFeed pour appeler bindTogglers

```
var tmr = window.setTimeout(function() {
    …
    $('results').update(adjustData(html));
    bindTogglers();
    hideIndicator();
}, 10);
```

Rechargeons et cliquons sur un lien de bascule (figure 9-6).



Figure 9–6
Un résultat de bascule… surprenant.

Mais qu'est-ce que c'est que ce charabia ? Eh bien, ne vous avions-nous pas dit que le HTML encodé était transmis en tant que nœud texte, et qu'il allait falloir l'interpréter nous-mêmes ? Voilà, nous y sommes. Retroussons nos manches.

Les mains dans le cambouis : interpréter le HTML encodé

Commençons par transformer ce HTML encodé en HTML décodé ; nous peaufinerons avec le formatage des dates W3DTF pour finir.

Traiter des quantités massives de HTML encodé

Le XHTML retourné par la transformation XSLT est à laisser globalement intact : seules les parties de contenu, en HTML encodé, sont à décoder. La taille totale du XHTML va bloquer les méthodes `replace` ou `gsub`, sans parler du `unescapeHTML` de Prototype, sur certains navigateurs. Nous allons donc devoir réaliser la recherche, le décodage et le remplacement « à la main ».

Voici le code dans `adjustData` qui va remplacer les fragments de HTML pour les contenus de billet par leur version décodée. Souvenez-vous qu'il y a deux `div` imbriqués dans le HTML produit : le `div` conteneur et un autre `div` simple en prévision d'un effet `SlideDown`.

Listing 9-22 La fonction adjustData avec le code de décodage ciblé

```
function adjustData(html) {
    DIV_OPENER = '<div class="content"';
    DIV_CLOSER = '</div>';
    var start = 0;
    var pos = html.indexOf(DIV_OPENER, start);
    var result = '';
    while (-1 != pos) {
        var openerEnd = html.indexOf('<div>', pos) + 5;
        result += html.substring(start, openerEnd);
        var closerStart = html.indexOf(DIV_CLOSER, openerEnd);
        result += safeUnescape(html.substring(openerEnd, closerStart));
        start = closerStart;
        pos = html.indexOf(DIV_OPENER, start);
    }
    result += html.substring(start);
    return result;
} // adjustData
```

Il s'agit de code de traitement de chaîne bête et méchant. Voyons maintenant notre fonction `safeUnescape`, qui remplace le `unescapeHTML` de Prototype afin de fonctionner même sur de gros textes quel que soit le navigateur.

Listing 9-23 La fonction safeUnescape, ou unescapeHTML garanti

```
function safeUnescape(text) {
    TRANSITIONS = { 'lt': '<', 'gt': '>', 'quot': '"',
        'apos': "'", 'amp': '&' };
    for (var entity in TRANSITIONS) {
        var re = new RegExp('&' + entity + ';&apos;, 'mg');
        text = text.replace(re, TRANSITIONS[entity]);
    }
    return text;
} // safeUnescape
```

Décidément, ces objets anonymes comme *hashes* sont bien pratiques. Ici, ils nous simplifient la déclaration des associations d'entités. Nous traitons les cinq entités encodées par XML (et donc HTML) : les chevrons, les guillemets simples et doubles, et l'esperluette (&). Les drapeaux `m` et `g` pour l'expression rationnelle assurent un remplacement total sur toutes les lignes.

Le résultat est présenté à la figure 9-7.

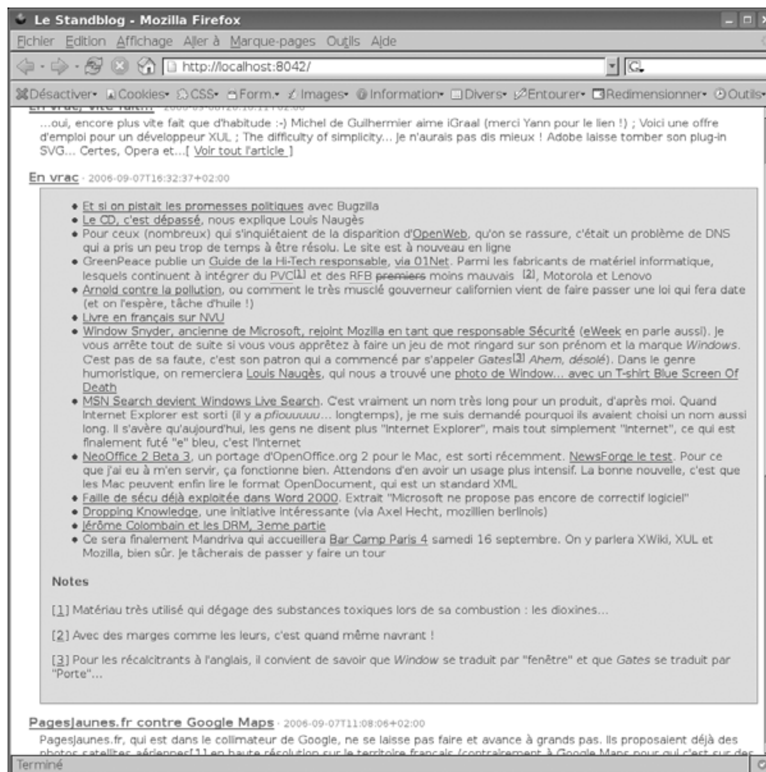


Figure 9–7
C'est beaucoup mieux !

Les dates W3DTF

Il ne nous reste plus qu'à traiter les dates W3DTF. Commençons par compléter `adjustData`.

Listing 9-24 Notre fonction adjustData, terminée (fragment)

```
function adjustData(html) {
    var html = html.gsub(RE_TIMESTAMP, function(match) {
        match[2] = w3dtfToDate(match[2]).toLocaleString();
        return match[1] + match[2] + match[3];
    });
    DIV_OPENER = '<div class="content"';
    …
} // adjustData
```

La fonction `w3dtfToDate` est un rien lourde... Nous utilisons une expression rationnelle pour découper le texte, un itérateur pour convertir les fragments numériques en objets `Number` et une série d'invocations sur un objet `Date` tout frais. Il faut en plus gérer le décalage GMT.

Une date W3DTF a le format suivant :

```
yyyy-mm-dd[Thh:mm[:ss](Z|(+-)hh:mm]
```

L'heure n'est pas obligatoire, et quand elle est là, elle ne précise pas forcément les secondes. Elle précise en revanche toujours un décalage GMT, soit avec `Z` (zéro décalage, donc GMT), soit sous format signé numérique. Dans notre expression rationnelle, nous considérons que l'heure et les secondes sont toujours là.

Pour prendre en compte le décalage, il faut stocker les données horaires comme s'il s'agissait de données GMT, puis appliquer un décalage opposé à celui indiqué. L'heure GMT stockée est ainsi correcte, et le formatage local, qui prend en compte *votre* fuseau horaire, affichera une heure dans *votre* référentiel.

Voici le code...

Listing 9-25 La fonction de conversion w3dtfToDate

```
RE_W3DTF = '^(\\d{4})-(\\d{2})-(\\d{2})T(\\d{2}):(\\d{2}):(\\d{2})
            ➥ ([+-]\\d{2}:\\d{2}|Z)$';

function w3dtfToDate(text) {
    var comps = $A(text.match(RE_W3DTF)).map(function(s, index) {
        return 0 == index || 7 == index ? s : parseInt(s, 10); ❶
    });
    var result = new Date();
    with (result) {
        setUTCFullYear(comps[1]);
        setUTCMonth(comps[2]);
        setUTCDate(comps[3]);
        setUTCHours(comps[4]);
        setUTCMinutes(comps[5]);
        setUTCSeconds(comps[6]);
    }
    var utcOffset = 0;
    if ('Z' != comps[7]) {
        var sign = '-' == comps[7].charAt(0) ? -1 : 1;
        var hours = parseInt(comps[7].substring(1, 3), 10);
        var minutes = parseInt(comps[7].substring(4), 10);
        utcOffset = sign * (hours * 60 + minutes);
    }
    result = new Date(result.getTime() - utcOffset * 60000);
    return result;
} // w3dtfToDate
```

L'exclusion des indices en ❶ repose sur le fait que l'indice 0 correspond à l'ensemble de l'expression (tout le texte W3DTF, qui ne nous est d'aucune utilité tel quel), et l'indice 7

est le décalage GMT, qui n'est jamais un simple nombre : nous voulons le conserver en tant que texte, pour analyse dans le bloc conditionnel (`if 'Z' != comps[7]...`).

Sauvons, rafraîchissons, et voici, comme sur la figure 9-8, de jolies dates en heure locale (qui est probablement la même que celle du Standblog : le fuseau horaire parisien).



Figure 9–8
Notre affichage de flux finalisé (texte grossi pour voir les dates/heures)

Pour aller plus loin...

Voilà la fin des chapitres, mais il vous reste beaucoup de choses très utiles à apprendre, découvrir ou redécouvrir dans les annexes. Croyez-le, elles ont autant de valeur que le reste du livre, et vous y apprendrez très probablement quelque chose.

Dans le cadre de ce chapitre précis, voici quelques références.

Livres

RSS et Atom : Fils et syndication
Heinz Wittenbrink
Eyrolles, 8 juin 2006, 295 pages
ISBN 2-212-11934-8

Tout ce que vous avez toujours voulu savoir sur les formats de syndication, de leur histoire aux micro-détails techniques, avec tous les modules RSS 1.0 et RSS 2.0. Atom et APP sont également documentés. Tout est là. Vraiment tout.

Sites

- La RFC 4287 : Atom 1.0 et *Atom Publishing Protocol*
  – http://tools.ietf.org/html/rfc4287
- La « spécification » RSS 2.0
  – http://blogs.law.harvard.edu/tech/rss
- La spécification RSS 1.0 et ses modules
  – http://web.resource.org/rss/1.0/
- La spécification XSLT
  – http://w3.org/TR/xslt/

Annexes

Annexe A

Bien baliser votre contenu : XHTML sémantique

Annexe B

Aspect irréprochable et flexible : CSS 2.1

Annexe C

Le « plus » de l'expert : savoir lire une spécification

Annexe D

Développer avec son navigateur web



Bien baliser votre contenu : XHTML sémantique

Au commencement était le contenu. La toute première étape de la création de votre page web doit être ce contenu. Il n'y a pas plus important. C'est lui qui fait la différence entre votre page et les autres. C'est ce contenu que voient avant tout les moteurs de recherche et les périphériques alternatifs. C'est lui qui définit la valeur, le sens et l'identité de votre page. Vous avez le devoir, je dis bien le devoir, de lui consacrer tout le soin possible.

Ce n'est qu'une fois votre contenu défini et mis en place que vous pouvez vous poser la question de son aspect et de son comportement. L'aspect sera le rôle des CSS, et nous verrons à l'annexe B que leur usage ne nécessite qu'extrêmement peu d'intrusions dans votre contenu pour satisfaire vos envies les plus variées. Quant au comportement, ce sera le rôle de JavaScript, et si vous devez retenir une chose des chapitres 2 et 3, c'est que son utilisation n'entache pas non plus votre contenu.

Cette annexe commence par présenter rapidement les nombreux bénéfices qu'apportent la présence de contenus « propres » dans vos pages web, c'est-à-dire, pour l'essentiel, des contenus valides et surtout sémantiques. Ensuite, vous trouverez un court passage en revue des contraintes syntaxiques, des éléments disponibles et de leur utilisation pertinente, pour finir par quelques exemples concrets.

Les avantages insoupçonnés

Prendre les bonnes habitudes pour créer un contenu de qualité ne rend pas le travail plus difficile (bien au contraire), mais apporte de nombreux bénéfices, parfois inattendus.

Pour le site et ses propriétaires

En tant que commanditaire d'un site web, qu'avez-vous à gagner à ce qu'il soit réalisé en XHTML 1 Strict, et à ce qu'il utilise un balisage rigoureusement sémantique ?

Beaucoup de choses :

1. Il sera infiniment plus compréhensible par les moteurs d'indexation et de recherche. Par conséquent, votre visibilité sur Google et autres, mais surtout la pertinence de cette visibilité, seront grandement améliorées.
2. Le balisage sémantique entraîne automatiquement une accessibilité accrue, ce qui signifie que votre site est consultable par un plus large public.
   Il s'agit d'abord des internautes atteints d'un handicap (visuel, moteur, cognitif), ce qui vous met en conformité avec les lois présentes ou à venir quant à l'accès aux sites web pour ces personnes.
   Il s'agit aussi des visiteurs utilisant autre chose qu'un ordinateur avec un grand écran, un clavier et une souris : ceux navigant sur votre site à l'aide d'un Palm, d'un PocketPC, d'un TabletPC, d'un téléphone 3G, d'un Blackberry, etc., ont par nécessité une préférence marquée pour les sites accessibles. Ils constituent qui plus est une cible marketing intéressante (adeptes de haute technologie et disposant de moyens financiers suffisant pour se les acheter).
3. Un tel site « pèse » beaucoup moins lourd en termes d'octets. D'innombrables refontes complètes de sites appliquant ces préceptes ont abouti à un contenu tout aussi riche mais beaucoup moins lourd. Conséquence directe : la consommation de bande passante (et donc son coût) diminue sensiblement, alors même que la fréquentation augmente (et donc les revenus aussi).
4. La fréquentation du site augmente, car il n'est plus nécessaire de maintenir différentes versions du contenu : l'utilisation des CSS permet de s'adapter automatiquement d'un périphérique de consultation à l'autre. Oublié, le temps où les versions alternatives (pour impression ou pour non-voyants) étaient souvent en retard sur la version principale ! Non seulement votre contenu est à jour pour tous, mais il devient accessible à une base plus large d'utilisateurs. Par ailleurs, cette centralisation du contenu signifie que sa mise à jour est plus simple et plus rapide : vous gagnez en parts, mais aussi en réactivité !
   Ce cercle vertueux ne peut que générer des revenus supplémentaires, qui ont tôt fait d'amortir les coûts d'une éventuelle refonte complète du site.

Vous retrouverez ces arguments sous un développement différent dans la section *À quoi servent les standards ?* de l'avant-propos. Il contient notamment un lien vers l'exemple célèbre du site de la chaîne de sports américaine ESPN, qui constitue un cas d'école magistral, avec des chiffres impressionnants à l'appui.

Pour le développeur web

Les développeurs chargés d'implémenter un site respectant ces principes de qualité ont aussi la vie plus facile.

1. Un document XHTML ne présente aucune ambiguïté, par comparaison à un document HTML classique. En y plaçant la bonne déclaration `DOCTYPE`, on peut tirer parti des validateurs en ligne pour vérifier la conformité du document à sa grammaire imposée, sans risque d'erreur. Des validateurs identiques existent pour les CSS, par exemple.
   Si les propriétaires du site imposent des règles éditoriales qui affectent la grammaire du document, il est possible de réaliser une DTD ou un schéma transcrivant ces contraintes pour bénéficier des mêmes tests automatisés de conformité.
2. Un des avantages de l'univers XML est qu'il permet, grâce au mécanisme des espaces de noms, de mélanger plusieurs vocabulaires dans un même document. Ainsi, en XHTML, on peut insérer des fragments de langages à balises supplémentaires, comme SVG pour les graphiques vectoriels, MathML pour les formules mathématiques ou encore SMIL pour le multimédia.
3. Un document sémantique a par définition beaucoup moins de balises qu'un document des années 1990, et beaucoup plus d'attributs `id` correctement définis : cela le rend plus simple à habiller avec les CSS, et plus simple à scripter en terme de DOM.

Règles syntaxiques et sémantiques

XHTML 1 Strict est la variante stricte de XHTML. XHTML lui-même peut se résumer à HTML 4.01 auquel on applique les règles syntaxiques de XML. Ces distinctions sont signalées dans la section 4 de la recommandation pour XHTML 1.0, et elles sont très simples.

- **Les documents doivent être correctement formés.** Cela signifie essentiellement que les balises doivent se fermer dans l'ordre inverse de leur ouverture : `<b><i>…</i>…</b>` est correct, mais `<b><i>…</b>…</i>` ne l'est pas. Les balises ne sont pas comme des commutateurs, mais constituent véritablement des conteneurs : on doit pouvoir déterminer quel élément est dans quel autre.

- **Par extension, tout élément doit être fermé.** En HTML, de nombreux éléments pourtant non vides n'étaient pas fermés par négligence : principalement li et option, mais aussi trop souvent p. On ferme un élément avec sa balise classique précédée d'un / : <p> devient </p>.
- **Les noms d'éléments et d'attributs doivent être en minuscules.** En XML, on est sensible à la casse, et les grammaires formelles pour XHTML utilisent les minuscules (c'est par ailleurs plus agréable visuellement).
- **Les éléments vides doivent aussi être fermés.** Il s'agit des éléments area, base, br, col, frame, hr, img, input, link, meta et param. Pour fermer un élément vide en XML, on ajoute simplement un / avant son chevron fermant. Toutefois, certains vieux navigateurs sont perturbés par cette notation ; aussi prend-on généralement la précaution d'ajouter une espace devant le /, comme ceci : .
- **Les valeurs d'attributs sont entre guillemets.** Techniquement, XML autorise tant les apostrophes (') que les guillemets ("). Je recommande vivement, à titre personnel, la deuxième possibilité.
- **Les attributs bascules ont tout de même une valeur.** En HTML, certains attributs sont obligatoires, mais sans qu'il soit nécessaire de préciser une valeur. C'est le cas notamment des attributs checked, compact, disabled, readonly et selected. En XML, un attribut a forcément une valeur. XHTML définit pour ces attributs une seule valeur autorisée, qui est leur propre nom. On écrira donc par exemple le code suivant : <option value="carrier" **selected="selected"**>Express</option>.
- Les navigateurs traitent les valeurs des attributs en supprimant tout espacement en début et en fin de valeur, et en réduisant toute série d'espacements interne à une seule espace.
- L'attribut name n'est plus autorisé pour les éléments suivants : a, applet, form, frame, iframe, img et map. En XML, le moyen standard d'identification d'un élément, quel qu'il soit, est l'attribut id. L'attribut name reste valide sur les autres éléments, notamment les champs de formulaires.
- Les valeurs par défaut des attributs sont fournies en minuscules uniquement (par exemple, get pour l'attribut method de form), toujours en raison de la sensibilité de XML à la casse.
- Un dernier détail : si vous utilisez des entités numériques avec un code hexadécimal, en XHTML le x initial est forcément en minuscules : … et non ….

La DTD et le prologue

On prendra également soin de toujours référencer la DTD en début de document, avant l'élément ouvrant `html`. Un contenu de qualité utilise soit la variante `Strict`, soit la variante `Frameset` si vous avez des cadres (frames). Toutefois, les cadres sont généralement considérés comme problématiques en termes d'accessibilité et de scripting. Ne les utilisez que si vous ne pouvez absolument pas faire autrement. Voici la déclaration `DOCTYPE` qui devrait figurer en haut de tous vos documents :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Cette déclaration n'a rien de superflu : sur la plupart des navigateurs, référencer une DTD stricte déclenche ce qu'on appelle le *doctype switching*, en forçant le navigateur à se placer en mode « respect des standards », ce qui change beaucoup de choses, notamment pour MSIE. D'ailleurs, toutes les améliorations CSS de MSIE 7 ne fonctionneront que dans ce mode, exigeant donc une déclaration de DTD stricte à l'ouverture du document.

Petit point de détail : un document XML (donc XHTML) utilisant un encodage autre que UTF-8 est censé le signaler d'entrée de jeu dans son prologue. Il s'agit d'une syntaxe spéciale présente dès le premier octet du document, et qui ressemble à ceci :

```
<?xml version="1.0" encoding="iso-8859-15"?>
```

Toutefois, MSIE 6 ne fonctionne pas correctement lorsqu'il reçoit un tel document servi comme du (X)HTML (ce point est corrigé dans MSIE 7). Aussi, on s'abstient pour l'instant de fournir un prologue. Heureusement, tous les navigateurs gèrent correctement la balise `meta`, qui fournit la même information, à condition que vous pensiez à la faire systématiquement figurer dans vos `head` :

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
15" />
```

La valeur initiale dépend d'ailleurs de la façon dont vous voulez « servir » votre contenu. Ici, c'est du HTML. On peut aussi opter pour le mode « XML », qui opère quelques petits changements à l'usage, ce qui affecte notamment le comportement du DOM. Le type MIME à utiliser alors est `application/xhtml+xml`. Choisir entre les deux et gérer les conséquences fait l'objet d'un débat incessant depuis de longues années...

XHTML, oui mais lequel ?

La version actuelle est XHTML 1.1, qui est toutefois encore loin d'être utilisée partout. Cette version n'a plus les variantes `Transitional` et `Loose` de la version 1.0 : elle considère qu'on écrit désormais du XHTML forcément strict (même si on conçoit des cadres) !

La version 1.1 date déjà : elle fut publiée le 31 mai 2001, soit à peine 16 mois après la publication initiale de la version 1.0. Fait plus marquant encore : la dernière révision de XHTML 1.0 date du 1er août 2002, elle est donc *plus* récente que XHTML…

Les différences avec XHTML 1.0, qui est de loin la variante la plus utilisée ces dernières années, sont très minimes, même pour une sous-version : l'attribut universel `lang` a été remplacé par `xml:lang` à des fins de compatibilité avec la modularisation en cours de XHTML, et une série d'éléments dits « ruby » a été ajoutée, qui permet de fournir des petits blocs de texte dans la marge, annotant le corps de texte principal.

Le gros du travail sur XHTML se concentre aujourd'hui sur XHTML 2.0, mais il s'agit là d'un des nombreux symptômes d'un malaise dans le W3C. En effet, l'immense majorité des ténors du Web ont vivement critiqué cette spécification, et estiment, pour résumer, que leurs auteurs se fourvoient.

Indiquons simplement qu'elle est énorme, divisée en pas moins de 26 modules (la tendance au W3C étant à la modularisation ces dernières années), certains étant minuscules (attributs noyau), d'autres énormes (XForms).

Les développeurs web et leurs spécialistes ont le sentiment que le comité n'a jamais eu à rédiger véritablement un site web, et travaille dans sa tour d'ivoire pour produire une recommandation que personne n'utilisera jamais…

Le balisage sémantique

C'est un concept très simple : il s'agit d'utiliser une balise pour son sens et non pas pour son aspect par défaut.

Encore aujourd'hui, vous trouverez nombre de développeurs web qui vous diront que « `h1`, c'est pour écrire en gros », ou que « `blockquote` sert à décaler vers la droite ». Ce genre de perception remonte aux premiers navigateurs, vers 1994. Autant dire, à l'échelle du Web, que ça remonte au Jurassique.

L'aspect par défaut d'une balise est sans aucune importance. Je dis bien : aucune. D'ailleurs, il n'existe aucun standard pour ces aspects par défaut (juste une suggestion en annexe D de la recommandation CSS 2.1). Vous ne trouverez pas un seul document formel disant à quoi doit ressembler une balise de titre, ni même `strong` ou `em`.

Bien sûr, on a longtemps eu des balises dédiées à l'aspect, par exemple `font`, `b`, `i`, `u` et `s`. Depuis HTML 4.01 (depuis 7 ans), seules les balises `b` et `i` sont encore autorisées, en raison de certains usages fréquents qui restent plus simples, et plus propres finalement, que de coller un `<span class="bold">…</span>` à la place.

Il est temps d'apprendre le véritable sens de ces balises. Ce faisant, vous découvrirez certaines balises et attributs dont vous n'aviez jamais entendu parler. Par exemple, connaissez-vous les balises `abbr`, `acronym`, `caption`, `cite`, `del`, `fieldset`, `ins`, `kbd`, `label`, `legend`, `q`, `samp`, `tbody`, `tfoot`, `th`, `thead` et `var` ? Je suis sûr qu'au moins certaines ne vous disent absolument rien. Et que dire des attributs `accesskey`, `axis`, `datetime`, `dir`, `for`, `lang`, `longdesc`, `scope` et `summary` ? Tout cela est pourtant lié à la sémantique et à l'accessibilité.

Ce n'est vraiment pas difficile de changer votre façon d'écrire du HTML. Commencez par aller découvrir les balises et attachez-vous à bien comprendre leur sens et leurs contextes autorisés d'utilisation. Lisez bien toute la description de la balise et de ses attributs. Fuyez toute balise et tout attribut dépréciés. La recommandation W3C est là pour ça, elle est claire, et HTML 4.01 dispose même d'une version française de bonne qualité (voir à la fin de cette annexe).

Les balises XHTML 1 Strict par catégorie

Cette section liste l'ensemble des balises autorisées en HTML 4.01 et donc en XHTML 1 (1.0/1.1, à l'exception du module Ruby de XHTML 1.1, trop peu implémenté, et des variantes `Transitional` et `Loose` de 1.0). Les balises dépréciées sont rapidement listées, dans une catégorie à part.

Dans les descriptions, vous trouverez une explication courte du sens et du contexte d'utilisation éventuel pour la balise, ainsi que quelques remarques, mises en garde, et surtout invitations à aller explorer certains attributs trop souvent ignorés ou mal employés.

Il ne s'agit pas de vous apprendre le HTML, pas plus que l'annexe 2 n'a pour ambition de vous apprendre CSS. Il y a d'excellents livres et sites pour cela ; vous en trouverez quelques-uns dans les sections *Pour aller plus loin...* de ces annexes. Il s'agit plutôt ici de vous rappeler les fondamentaux, et de vous ouvrir les yeux sur le reste.

Balises structurelles

Les balises structurelles sont celles qui décrivent la structure du document : en-têtes, corps, regroupement d'éléments en ligne ou de type bloc, tableaux...

Tableau A–1 Balises structurelles de XHTML 1

Balise	Description
body	Corps du document. Ne peut contenir que des éléments de type bloc, script, ins ou del. Les attributs background, text, link, vlink et alink ont été dépréciés en faveur des CSS.
col	Décrit le format d'une colonne de tableau. Présente dans table ou colgroup.
colgroup	Alternative de description des formats d'un groupe de colonnes. Présente dans table.
div	Conteneur de type bloc utilisé pour grouper d'autres éléments à des fins de CSS ou de script. Ne sombrez pas inutilement toutefois dans la *divitis* (http://fr.wikipedia.org/Divitis) !
head	En-têtes du document : titre, métadonnées, liens vers des feuilles de styles, scripts et documents connexes.
html	L'élément racine de tout document (X)HTML. Pour être impeccable, on renseigne ses attributs xmlns et xml:lang. comme indiqué dans la section 3.1.1 de la recommandation pour XHTML 1.0.
script	Permet de charger un script séparé (usage correct, dans le cadre de head) ou de définir un script interne. Ce dernier usage lui vaut sa place ici, mais il va à l'encontre des principes de l'*unobstrusive JavaScript*, décrits au chapitre 2. Voir également le tableau A-3.
span	Conteneur servant à regrouper des éléments de type *inline* à des fins de CSS ou de script.
table	Tableau de données. Ne jamais utiliser pour obtenir une mise en page ! Connaissez-vous son attribut summary ?
tbody	Conteneur de tout ou partie des lignes du corps d'un tableau, présent dans table après d'éventuels thead et tfoot. Oubliez les tr directement dans table, ce n'est pas soigné ! Lorsqu'un tableau contient plusieurs sous-sections logiques, il est bon de consacrer un tbody à chacune.
td	Cellule de tableau contenant une donnée (*Table Data*). Les attributs abbr, headers et scope font des merveilles pour l'accessibilité à l'intention des logiciels de lecture d'écran ou des personnes souffrant d'un handicap cognitif.
tfoot	Pied de tableau. Présente dans table après un éventuel thead, mais avant tbody. Censée être répétée en bas de chaque portion visible du tableau quand celui-ci est imprimé sur plusieurs pages, par exemple.
th	Cellule de tableau contenant un titre de ligne ou de colonne (*Table Heading*). Même remarque que pour td.
thead	En-tête de tableau. Présente dans table comme premier élément fils (exception faite de caption). Censée être répétée en haut de chaque portion visible du tableau quand celui-ci est imprimé sur plusieurs pages, par exemple.
tr	Ligne de cellules dans une portion de tableau (*Table Row*). Peut contenir des éléments th ou td.

Balises sémantiques

Les balises sémantiques forment la plus grande catégorie, pour la simple raison que HTML est conçu pour indiquer au mieux la signification des portions de texte qui forment son contenu. Si ce tableau ne vous fait pas découvrir un seul élément, chapeau bas !

Tableau A–2 Balises sémantiques de XHTML 1

Balise	Description
abbr	Indique une abréviation. Si on cherche l'exactitude, une abréviation est soit la version raccourcie d'un mot (par exemple « Mass. » pour « Massachussets »), soit une série d'initiales n'étant pas conçue pour être prononcée, mais plutôt pour être épelée (WWW, URI, HTTP, SNCF...).
acronym	Indique un acronyme, c'est-à-dire une série d'initiales conçue pour être prononcée (Wysiwyg, Radar, Adullact...).
blockquote	Bloc de citation, approprié pour une citation longue. L'attribut cite permet de fournir l'URI de la source. Voir aussi q.
caption	Titre d'un tableau, préférable de loin à une première tr dans thead avec un th doté d'un éventuel attribut colspan. Élément fils immédiat de table. Généralement affiché par défaut au-dessus du tableau (réglable par CSS), centré sur la largeur de celui-ci.
cite	Référence à une source externe (citation au sens de « citer ses sources »). Voir aussi blockquote et q.
code	Fragment de code informatique (dans un langage quelconque). Voir aussi kbd, samp et var.
dd	Corps d'une définition associée au dernier terme (dt) la précédant dans une liste de définition (dl). Un même terme peut avoir plusieurs définitions successives, ou plusieurs fragments de définition, sous forme de plusieurs éléments dd.
del	Indique une excision (retrait) de contenu dans le cadre d'une révision du document. Un des éléments les plus sous-utilisés de HTML. L'attribut datetime permet d'horodater la modification, et l'attribut cite peut référencer l'URI d'un document détaillant ses motivations. L'affichage par défaut est explicite, en laissant son contenu visible mais barré. Voir aussi ins.
dfn	La balise la plus mal spécifiée de tout HTML. Censée encadrer la première occurrence d'un terme, à valeur de définition. Mais la recommandation n'explique pas comment isoler le mot dans sa définition...
dl	Liste de définitions. Contient une série de termes (dt) suivis d'une ou plusieurs définitions (dd). La sémantique autorisée est un peu plus large que les définitions à proprement parler, de sorte qu'on peut s'en servir pour un glossaire, un menu, voire même des dialogues dans un scénario.
dt	Terme à définir, dans le cadre d'une liste dl. Suivie d'une ou plusieurs définitions dd.
em	Emphase simple, plus légère que strong. Dans un texte, correspond souvent à l'effet produit par la mise en italique.
h1...h6	Titres à niveaux hiérarchiques, h1 étant le premier niveau (le plus haut), h6 le dernier (on en a rarement besoin, ou alors votre page est un immense pavé). Les moteurs de recherche leur accordent en général un « poids » proportionnel.
img	Insère une image. Devrait toujours avoir un attribut alt, vide uniquement si l'image est purement décorative (pour ne pas gêner la représentation textuelle). Les attributs longdesc et title ont aussi un rôle à jouer dans l'accessibilité.
ins	Symétrique à del : indique un ajout de contenu dans le cadre d'une révision du document. Un des éléments les plus sous-utilisés de HTML. L'attribut datetime permet d'horodater la modification, et l'attribut cite peut référencer l'URI d'un document détaillant ses motivations.
kbd	Indique un texte saisi par l'utilisateur dans le cadre d'une manipulation. Voir aussi samp.

Tableau A–2 Balises sémantiques de XHTML 1 (suite)

Balise	Description
label	Indique un libellé de champ dans un formulaire. Figure aussi au tableau A-6 pour l'aspect formulaire. Ses attributs accesskey et for sont primordiaux, car certains types de champs ne peuvent définir leur propre accesskey. Attention : for référence un id, pas un name ! Peut se présenter sous deux formes, mais on préfère qu'il n'encadre que le libellé lui-même, et pas le champ : cela ouvre davantage de possibilités tant pour les CSS que pour les scripts.
li	Élément de liste (ul ou ol). Pensez à bien les fermer en XHTML !
ol	Liste ordonnée (séquence croissante de numéros ou lettres). Tous les attributs de séquencement (start, value) et de format (type, compact) ont été dépréciés au profit des CSS, plus puissantes d'ailleurs.
p	Définit un paragraphe, ce qui est en soi un type sémantique de contenu. Pensez à bien les fermer en XHTML !
q	Citation courte. L'attribut cite permet de référencer l'URI de la source. Voir aussi blockquote.
samp	Exemple d'un affichage ou d'une sortie imprimée par un programme (affichage en console, etc.). Voir aussi kbd.
strong	Emphase forte, plus appuyée que em. Dans un texte, correspond à l'effet produit par la mise en gras.
sub	Texte en indice, généralement dans une formule (*subscript*).
sup	Texte en exposant, généralement dans une formule ou pour une note de bas de page (*superscript*).
title	Titre de la page, présente dans head. Contient souvent au moins l'équivalent de l'éventuel élément h1 présent dans le corps de la page, mais avec plus de contexte. Son contenu constitue souvent le titre de la fenêtre de l'onglet du navigateur.
ul	Liste non ordonnée (à puces). Les attributs type et compact ont été dépréciés au profit des CSS, plus puissantes d'ailleurs. Idéal pour représenter sémantiquement un menu, les CSS pouvant le transformer à volonté…
var	Indique une variable ou un argument de programme. Voir aussi code et kbd.

Balises de liaison

Il s'agit de balises établissant des liens vers d'autres ressources ou documents.

Tableau A–3 Balises de liaison de XHTML 1

Balise	Description
a	La balise de lien par excellence. Attention : son attribut target est déprécié depuis déjà 7 ans, et l'attribut name n'est plus autorisé en XHTML : utilisez plutôt un id sur l'élément vers lequel vous souhaitez définir une URL de fragment. Côté sémantique, l'utilisation de l'attribut hreflang est en pleine explosion (notamment grâce aux sélecteurs CSS 2), l'attribut rel commence à faire surface grâce à des fonctionnalités proposées par Google, et l'attribut type peut être utile.
base	Modifie l'URL de base pour les URL relatives au sein du document. Présente dans head. Son attribut target permet, le cas échéant, de faire que tous les liens s'ouvrent dans un autre cadre que le cadre courant.

Tableau A–3 Balises de liaison de XHTML 1 (suite)

Balise	Description
link	Lien vers une ressource externe. Principal moyen de chargement de CSS. Ses attributs type et href sont incontournables, mais l'attribut rel est aussi très utile, tant pour proposer plusieurs feuilles de styles alternatives que pour définir des hiérarchies ou réseaux de pages. Connaissez-vous ses valeurs start, next, prev, contents, chapter ou encore section ? Il y en a bien d'autres : http://www.w3.org/TR/html401/types.html#type-links. Enfin, l'attribut hreflang existe ici aussi, ce qui est sémantiquement intéressant.
script	Chargement de scripts externes depuis le head. Seule utilisation correcte, contrairement aux fragments de scripts intégrés au contenu de la page (à l'exception des retours Ajax, comme cela est montré dans plusieurs chapitres). Attention : l'attribut language est déprécié depuis 7 ans, au profit de l'attribut type, qui vaut donc le plus souvent text/javascript.

Balises de métadonnées

Tableau A–4 Balises de métadonnées de XHTML 1

Balise	Description
address	Encadre les informations de contact des responsables de la page (liens mailto:, paragraphes avec une adresse postale,etc.). Extrêmement peu utilisé. À tort ?
bdo	Court-circuite l'algorithme déterminant le sens du texte (gauche à droite ou droite à gauche) sur la base des attributs dir des éléments contenant celui-ci (*Bi-Directional Override*). Son attribut dir redéfinit la direction par défaut de son contenu textuel, tandis que son attribut lang (ou xml:lang en XHTML 1.1) permet d'en changer aussi la langue indiquée. J'avoue ne pas percevoir l'intérêt par rapport à span, qui n'a pas plus de valeur sémantique et peut faire la même chose...
meta	L'élément de métadonnées par excellence. Figure dans head, autant de fois que nécessaire. Peut aussi bien remplacer ou compléter les en-têtes de réponse HTTP (attribut http-equiv, très utilisé notamment pour préciser le jeu de caractères grâce à Content-Type) que rajouter des métadonnées quelconques (attribut name). La valeur est dans l'attribut content. On l'utilise notamment pour remplir les pages de métadonnées appartenant à des ontologies (vocabulaires) reconnues, comme le Dublin Core (http://dublincore.org/documents/dcmi-terms/). Connaissez-vous son attribut scheme, plutôt avancé ?

Balises de présentation

Tableau A–5 Balises de présentation de XHTML 1

Balise	Description
area	Définit une zone cliquable sur une image à l'aide de ses attributs shape, coords et href. Ne négligez surtout pas ses attributs alt, tabindex et accesskey pour l'accessibilité ! Voir aussi map.
b	Mise en gras. Jugé plus soigné qu'un … en l'absence de connotation sémantique.

Tableau A–5 Balises de présentation de XHTML 1 (suite)

Balise	Description
`big`	Utilisation de la taille de texte supérieure. J'avoue ne pas en voir l'utilité par rapport aux CSS. Voir aussi `small`.
`br`	Fin de ligne, ou retour chariot si vous préférez (*line break*). Force le passage à la ligne dans un texte.
`frame`	Définition d'un cadre dans un ensemble de cadres. Je rappelle que les cadres sont à éviter le plus possible, car ils nuisent à l'accessibilité et à la navigation. Si vous en utilisez néanmoins, assurez-vous de bien définir l'attribut `longdesc`.
`frameset`	Définition d'un ensemble de cadres, qui précise notamment la disposition (horizontale ou verticale).
`hr`	Ligne horizontale de séparation (*horizontal rule*). Tous ses attributs spécifiques sont dépréciés au profit des CSS.
`i`	Mise en italique. Jugé plus soigné qu'un `<span class="italic">…</span>` en l'absence de connotation sémantique.
`map`	Conteneur descriptif pour les zones cliquables (`area`) d'une image (`img` ; je vous déconseille d'utiliser des zones cliquables sur un `input type="image"` ou un `object`, même si c'est autorisé). Son attribut `name` est référencé par l'attribut `usemap` de l'image. Attention à bien renseigner les attributs d'accessibilité des balises `area`.
`noframes`	Contenu alternatif pour un navigateur ne prenant pas en charge les cadres. Principalement utile pour les moteurs de recherche, qui n'iront pas dans les cadres. En revanche, tous les navigateurs répandus, même textuels (w3m a supplanté lynx), gèrent aujourd'hui les cadres.
`object`	Balise fourre-tout pour objets ne disposant pas d'une balise dédiée : applets Java (la balise `applet` est dépréciée depuis 7 ans), animations Flash ou Shockwave, et sur MSIE n'importe quel contrôle ActiveX (par exemple Windows Update). Essentiellement utilisé pour fournir des animations et des lecteurs audio ou vidéo au sein de la page.
`param`	À l'intérieur d'un `object`, les éléments `param` permettent de définir des paramètres nommés pour l'objet.
`pre`	Texte préformaté : il s'agit généralement de l'équivalent d'un `<div style="white-space: pre;">`. On y laisse donc normalement les espacements tels quels, ainsi que les retours chariot : les lignes ne sont pas découpées pour satisfaire une contrainte de largeur. Très utilisée pour présenter du code source ou tout autre contenu textuel de nature informatique. La police de caractères par défaut est à chasse fixe (par exemple `Courier New`).
`small`	Utilisation de la taille de texte inférieure. J'avoue ne pas en voir l'utilité par rapport aux CSS. Voir aussi `big`.
`style`	Permet de fournir un fragment de script dans le corps du document. Cette utilisation est à proscrire, car elle est contraire à la séparation du contenu et de l'aspect. Ne pas confondre avec l'attribut `style` de la plupart des éléments, qui est parfois justifié dans certains contextes très délimités. Voir aussi `link` dans le tableau A-3.
`tt`	Utilisation d'une police à chasse fixe (*teletype*). À n'utiliser que si vous estimez que votre contenu n'est pas couvert sémantiquement par `code`, `kbd`, `samp` ou `var`.

Balises de formulaires et d'interaction

La plupart des champs de formulaires et boutons disposent d'attributs relatifs à l'ergonomie et l'accessibilité, qu'il vous faut bien connaître. Citons notamment `accesskey` (qu'il est préférable de laisser toutefois sur un `label` associé), `disabled`, `readonly` et `tabindex`.

Tableau A–6 Balises de formulaire et d'interaction de XHTML 1

Balise	Description
`button`	Variante peu utilisée (car souvent visuellement… bizarre) de `<input type="button"… />`. La balise `button` est en fait un conteneur, dans lequel on peut placer tout type de contenu en ligne, à l'exception des balises de formulaire et de `a`. Elle est donc censée permettre un contenu de bouton plus riche que `input`. Personnellement, je ne m'en sers jamais.
`fieldset`	Groupe logique de champs dans un formulaire, doté le plus souvent d'un titre grâce à `legend`. Encore trop peu utilisé et pourtant très utile pour des formulaires un peu voire très riches en contenu. Vous verrez un exemple plus loin dans cette annexe.
`form`	Formulaire. Grammaticalement, il n'est pas possible de placer un champ de formulaire hors d'une balise `form`, quitte à ce qu'elle soit inutile par ailleurs. Les principaux attributs sont `method` et `action`. Si vous avez des champs de type fichier, l'attribut `enctype` est indispensable. Quant à `name`, il est déprécié depuis XHTML 1.0, au profit de `id`. Par ailleurs, la DTD exige que `form` ne contienne que des éléments de type bloc ou `script`, hormis `form` bien sûr. Conséquence : vos champs de formulaire doivent apparaître dans des conteneurs de type bloc, souvent `p` ou `fieldset`.
`input`	La principale balise de description de champ. Son attribut `type`, quoique doté d'une valeur par défaut (`text`), devrait toujours être précisé, ainsi que `name` (sauf pour le type `submit`, à moins qu'il y en ait plusieurs) et `tabindex`. Pensez aussi, quand c'est pertinent, aux attributs `accept`, `checked`, `disabled`, `maxlength`, `readonly` et `value`. Quant à `size`, préférez utiliser une règle CSS.
`label`	Libellé d'un champ de formulaire. Signalé au tableau A-2 pour son rôle sémantique, ce pour quoi je vous encourage bien entendu à fournir un libellé pour tout champ n'en ayant pas déjà un (donc, tous sauf les boutons). Pensez aussi à associer explicitement le `label` à son champ à l'aide d'un `id` sur le champ et du `for` correspondant sur le `label`. `select` ne disposant pas d'attribut `accesskey`, `label` permet de lui en associer un ; d'une manière générale, mettez les `accesskey` sur les libellés quand ils existent.
`legend`	Titre d'un groupe logique de champs et libellés dans un formulaire. Généralement le premier élément fils d'un `fieldset`.
`optgroup`	Dans un `select`, définit un groupe d'options possibles, ce qui est pratique quand il y en a beaucoup. On utilise l'attribut `label` pour nommer le groupe, et on peut même désactiver juste ce groupe avec `disabled`. On ne peut pas imbriquer les éléments `optgroup` : la hiérarchie n'a qu'un niveau. Suivant le navigateur, l'aspect par défaut varie mais un style peut bien entendu être appliqué.
`option`	Option sélectionnable dans un `select`. Peut figurer directement sous le `select` ou dans un `optgroup`. L'attribut `selected` parle de lui-même… Désactivable individuellement avec l'attribut `disabled`. Il est préférable de toujours préciser l'attribut `value`, le comportement normatif (utilisation du contenu textuel comme valeur par défaut) n'étant pas respecté au niveau DOM par MSIE.

Tableau A–6 Balises de formulaire et d'interaction de XHTML 1 (suite)

Balise	Description
`script`	Permet l'inclusion de fragments de script directement dans le corps du document, principalement pour manipuler un fragment DOM juste après sa création, sans attendre le chargement complet de la page. C'est une pratique qui va plutôt à l'encontre de l'*unobstrusive JavaScript*, aussi je la déconseille fortement. Là encore, il existe quelques cas, notamment la récupération de fragments XHTML + JS en Ajax, où cette utilisation est toutefois acceptable.
`select`	Liste d'options, soit déroulante (attribut `size` de valeur `1`), soit dépliée mais avec un nombre d'options visibles fixe (valeur de l'attribut `size`). Peut contenir des balises `option` et `optgroup`. L'attribut `multiple`, s'il est fourni, indique que l'utilisateur peut sélectionner plusieurs éléments. Attention : il est ergonomiquement incompatible avec le mode déroulant.
`textarea`	Zone de saisie de texte multiligne. Utilise une balise fermante. Attention : tout le contenu entre la fin de la balise ouvrante et le début de la fermante fait partie de la valeur, espacements et retours chariot compris ! Les attributs importants pour l'aspect par défaut sont `cols` et `rows`. MSIE y ajoute une pléthore d'extensions, comme pour tous les champs de formulaire, mais la plus fréquemment rencontrée est l'attribut `wrap`. Évitez tous ces ajouts non standards.

Balises dépréciées

Voilà déjà 7 ans (HTML 4.01, décembre 1999) que les balises dépréciées n'ont plus droit de cité. Et pourtant, on les rencontre encore souvent... hélas ! Évitez donc de vous attirer les foudres des navigateurs modernes et de vos pairs.

Tableau A–7 Balises dépréciées de XHTML 1

Balise	Description
`applet`	Permettait l'inclusion d'une applet Java. Remplacée par `object`.
`basefont`	Permettait de préciser les valeurs par défaut pour les attributs des balises `font`.
`center`	Centrait tout son contenu. Remplacée par les propriétés CSS `text-align` et `margin` (valeur spéciale `auto`). Attention : l'attribut `align` est également déprécié pour tous les éléments !
`dir`	Devait permettre de créer des listes à colonnes multiples, comme un annuaire (*directory*). N'a jamais été vraiment prise en charge. Voir aussi `menu`.
`font`	Remplacée par les CSS, bien plus puissantes d'ailleurs.
`iframe`	Sorte de cadre en ligne, qui était abondamment utilisé (et, hélas, l'est encore) pour réaliser des échanges en arrière-plan avec le serveur (façon Ajax), ce qui était au moins autant utilisé à des fins légitimes qu'à d'autres bien plus dangereuses. Pas véritablement dépréciée, mais qui ne figurait déjà plus que dans la variante `Loose` de HTML 4.01, la moins soignée.
`isindex`	Très vieille variante de champ de saisie textuelle à ligne unique.
`menu`	Devait permettre de créer des listes à colonne unique. N'a jamais été vraiment prise en charge. Voir aussi `dir`.
`s`	Synonyme de `strike`.

Tableau A–7 Balises dépréciées de XHTML 1 (suite)

Balise	Description
`strike`	Barrait un contenu. Remplacée par `del` quand la sémantique convient, par la propriété CSS `text-decoration` sinon.
`u`	Soulignait le texte. Remplacée par la propriété CSS `text-decoration`.

Attributs incontournables

Pour finir, voici quelques attributs incontournables, dits « noyau », identifiables dans la DTD sous l'entité `%coreattrs`. Tous les éléments en disposent.

Tableau A–8 Attributs incontournables de XHTML 1

Attribut	Description
`class`	Contient une ou plusieurs classe(s) CSS. Le saviez-vous ? Il est tout à fait possible de fournir plusieurs classes à un même élément, qui cumule alors les applications de règles. Cela est très pratique, et évite d'avoir à dupliquer nombre de règles dans la feuille de styles. Les classes sont séparées dans la valeur de l'attribut par des espaces.
`dir`	Indique la direction d'écriture du contenu (`ltr` ou `rtl`, pour gauche à droite ou droite à gauche). Voir aussi `lang`.
`id`	De loin l'attribut le plus utilisé, avant même `class`, dans un document soigné et sémantique ! Contient un identifiant, unique à travers tout le document, grâce auquel tant les CSS que les scripts pourront référencer l'élément en un clin d'œil. Le saviez-vous ? Un ID valide doit commencer par une lettre non accentuée et peut ensuite contenir des chiffres arabes et des tirets bas (_), mais aussi des tirets (-), des points (`.`) et des deux-points (`:`) ! Ces deux dernières possibilités peuvent toutefois poser un problème dans la syntaxe des règles CSS, qui ont alors besoin d'encadrer les ID par des apostrophes (`'`) pour s'y retrouver.
`lang`	Langue du contenu. Contient un code de langue de la RFC 1766, par exemple `fr-FR`. Voir aussi `dir`.
`style`	Style en ligne appliqué à l'élément. Généralement mal vu, car enfreint la séparation entre contenu et aspect. Parfois nécessaire cependant, en particulier pour masquer initialement un contenu avec `display: none` dans le cadre des bibliothèques Prototype et donc script.aculo.us (voir chapitres 4, 6 et 7).
`title`	Fournit une information descriptive ou complémentaire sur l'élément. L'information est généralement affichée dans une infobulle si le curseur de la souris reste un instant sur l'élément. Fondamental pour des balises comme `abbr` et `acronym`, il est aussi utile pour préciser la destination d'un lien ou nommer les feuilles de styles alternatives d'un document. Il a enfin de nombreux usages dans le cadre de l'accessibilité.

Besoins fréquents, solutions concrètes

À présent que nous avons fait le tour des balises disponibles en XHTML 1 Strict, il est temps d'en illustrer l'utilisation la plus correcte possible dans le cadre de certains besoins récurrents.

Un formulaire complexe mais impeccable

Prenons un formulaire sur un site communautaire, servant à éditer des informations de profil. Les concepteurs du site ont décidé que la totalité des informations étaient présentées dans un même formulaire. Ces informations sont réparties en 5 groupes logiques :

1. connexion (identifiant, mot de passe) ;
2. identité (civilité, nom, prénom, surnom) ;
3. détails personnels (adresse courriel, identifiants de messagerie instantanée, site personnel, petit descriptif en texte libre, photo) ;
4. centres d'intérêts (série de cases à cocher) ;
5. conditions de fonctionnement (CGU, divulgation d'informations à des partenaires, type d'abonnement à la lettre de diffusion du site).

Le formulaire doit être le plus lisible possible, pour tout type de public. On veillera donc à éviter une mise en page tableau, pour lui préférer l'emploi de CSS adaptables au type de navigateur (petit ou grand écran, graphique ou texte, lecteur d'écran ou plage braille, etc.) et aux besoins de l'utilisateur (navigation intelligente à l'aide de la touche `Tab` et de touches de raccourci, libellés associés aux champs, etc.).

La photo peut être téléchargée à l'aide d'un champ de type fichier. Ce champ doit filtrer les possibilités pour ne retenir que les formats d'image pris en charge par le site, pour lequel on a décidé de se limiter à JPEG, PNG et GIF. Une limite de taille de fichier sera gérée côté serveur, et l'image y sera de toutes façons redimensionnée.

Évidemment, toute validation ou complément fonctionnel JavaScript sera réalisé de façon non intrusive, depuis un script séparé. Aucun gestionnaire d'événement ne sera donc présent dans le code, et un envoi sans activation JavaScript sera possible (avec un champ de type `submit` ou `image`).

Voici le code HTML répondant à ce besoin (vous le retrouverez dans l'archive de codes source pour ce livre, disponible sur le site des éditions Eyrolles).

Listing A-1 Notre première section, dédiée aux informations de connexion

```
<form id="profile" enctype="multipart/form-data"
      ➥ method="post" action="/profile/update">
```

```
<fieldset id="credentials">
    <legend>Connexion</legend>
    <p>
        <label for="edtReqLogin" accesskey="E">Identifiant</label>
        <input type="text" id="edtReqLogin" name="login"
               ➥ value="tdd" tabindex="1" />
    </p>
    <p>
        <label for="edtPassword" accesskey="A">Nouveau mot de passe
        </label>
        <input type="password" id="edtPassword" name="password"
               ➥ tabindex="2" />
    </p>
    <p>
        <label for="edtConfirm" accesskey="C">Confirmation</label>
        <input type="password" id="edtConfirm" name="confirm"
               ➥ tabindex="3" />
    </p>
</fieldset>
```

- Notez l'attribut `enctype` du formulaire `form`, en prévision du champ de téléchargement de fichier du listing A-3.
- Chaque champ fait l'objet d'une valeur à progression logique pour `tabindex`.
- Le mot de passe étant saisi sans confirmation visuelle, on ajoute un champ de confirmation pour détecter une saisie erronée.
- Tous les libellés définissent une `accesskey` pour leur champ.

Voyons à présent la deuxième section du formulaire.

Listing A-2 Informations sur l'identité de l'utilisateur

```
<fieldset id="identity">
    <legend>Identité</legend>
    <p>
        <label for="cbxTitle" accesskey="V">Vous êtes</label>
        <span id="basicIdentity">
          <select id="cbxTitle" name="title" size="1" tabindex="4">
                <option value="1" selected="selected">M.</option>
                <option value="2">Mlle</option>
                <option value="3">Mme</option>
           </select>
          <input type="text" id="edtReqFirstName" name="firstName"
                ➥ accesskey="P" value="Christophe" tabindex="5" />
           <input type="text" id="edtReqFirstName" name="lastName"
                ➥ accesskey="N" value="Porteneuve" tabindex="6" />
        </span>
    </p>
```

```
        <p>
            <label for="edtNickname" accesskey="S">Surnom</label>
            <input type="text" id="edtNickname" name="nickname"
                   ➥ value="TDD" tabindex="7" />
        </p>
    </fieldset>
```

- Afin de gagner en place et en ergonomie, nous plaçons les champs composant l'identité globale (civilité, prénom, nom) sur une même ligne. Pressentant un besoin de groupement pour la CSS, on encadre les trois par un `span`.
- Le `select`, en `size` 1, est une liste déroulante. L'élément retenu par défaut est marqué par un attribut bascule `selected`. Notez sa valeur, seule autorisée par la DTD.

La section sur les détails personnels est de loin la plus riche.

Listing A-3 Détails personnels

```
  <fieldset id="details">
    <legend>Détails personnels</legend>
    <p>
      <label for="edtReqEmail" accesskey="O">Courriel</label>
      <input type="text" id="edtReqEmail" name="email"
             ➥ value="tdd@example.com" tabindex="8" />
    </p>
    <p>
      <label><abbr title="Instant Messaging,
                   ➥ Messagerie instantanée">IM</abbr></label>
      <span id="imIDs">
        <label for="edtMSN" accesskey="1">MSN</label>
        <input type="text" id="edtMSN" name="im[msn]"
               ➥ value="tdd@example.com" tabindex="9" title="Alt+1" />
        <label for="edtICQ" accesskey="2">ICQ</label>
        <input type="text" id="edtICQ" name="im[icq]"
               ➥ value="123456789" tabindex="10" title="Alt+2" />
        <label for="edtJabber" accesskey="3">Jabber</label>
        <input type="text" id="edtJabber" name="im[jabber]"
               ➥ value="tdd@example.com" tabindex="11" title="Alt+3" />
      </span>
    </p>
    <p>
      <label for="edtHomePage" accesskey="T">Site personnel</label>
      <input type="text" id="edtHomePage" name="homePage"
             ➥ value="http://www.example.com" tabindex="12" />
    </p>
    <p id="detailsContainer">
      <label for="memDetails" accesskey="D">Détails</label>
      <textarea id="memDetails" name="details" cols="40" rows="3"
                ➥ tabindex="13"></textarea>
    </p>
```

```
    <p>
      <label for="filPhoto" accesskey="H">Photo</label>
      <span id="photo">
        <input type="file" id="filPhoto" name="photo" tabindex="14"
                ➥ accept="image/jpeg,image/gif,image/png" />
        <span class="note">(JPEG, PNG ou GIF, 150ko maximum)</span>
      </span>
    </p>
  </fieldset>
```

- Notez l'explication de l'abréviation IM, à l'aide d'`abbr` et de son attribut `title`. La CSS incitera l'utilisateur à amener sa souris sur le terme et à l'y laisser un instant pour obtenir l'explication.
- Là aussi, quelques éléments sur la même ligne sont regroupés dans un `span` en prévision des CSS.
- Remarquez comme les champs de saisie d'identifiants IM précisent leur touche de raccourci avec l'attribut `title` : le caractère actif ne figurant pas dans leur libellé, il ne pourra pas être mis en évidence par un script. Il faut tenter de fournir l'information autrement, et `title` produit généralement une infobulle.
- Remarquez que `textarea` est un élément non vide : il faut utiliser une balise fermante. Les suggestions de taille proposées par les attributs `cols` et `rows` seront remplacées par celles de la CSS quand celle-ci est active.
- Exemple de champ de type `file`. Il n'est pas possible de spécifier une valeur dans le HTML. Notez l'attribut `accept`.

La section 4, celle des centres d'intérêt, est plus simple. On suppose que la liste de ceux-ci est dynamiquement extraite de la base de données, ce qui empêche l'utilisation de touches de raccourci, car on ne connaît pas forcément tous les libellés affichés.

Listing A-4 Centres d'intérêts

```
    <fieldset id="interests">
        <legend>Centres d'intérêts</legend>
        <ul>
            <li>
                <input type="checkbox" id="chkInterests_1"
                        ➥ name="interests" value="1" />
                <label for="chkInterests_1">Jeux vidéo</label>
            </li>
            <li>
                <input type="checkbox" id="chkInterests_2"
                        ➥ name="interests" value="2" />
                <label for="chkInterests_2">Mode</label>
            </li>
```

```
            <li>
                <input type="checkbox" id="chkInterests_3"
                       ➥ name="interests" value="3" />
                <label for="chkInterests_3">Gadgets</label>
            </li>
        </ul>
    </fieldset>
```

Ici, une liste non ordonnée est sémantiquement parfaite pour représenter cette liste de choix individuels. La CSS fera disparaître les puces et ajustera les positions. Pour faciliter le traitement côté serveur, on donne à tous les champs le même nom, mais une valeur différente (ici probablement l'ID de l'information dans la base), utilisée pour composer l'`id` HTML et ainsi associer le libellé à la case. Cette association permet à l'utilisateur de basculer l'état de la case en cliquant sur le libellé lui-même, ce qui est plus accessible et similaire aux interfaces classiques. Côté serveur, on recevra le champ une fois par case cochée, avec la bonne valeur à chaque fois.

Enfin, voici la dernière section, le bouton d'envoi et la clôture du formulaire.

Listing A-5 Conditions de fonctionnement

```
    <fieldset id="account-behavior">
        <legend>Conditions de fonctionnement</legend>
        <ul>
            <li>
                <input type="checkbox" id="chkCGU" name="cgu"
                       ➥ value="yes"checked="checked" />
                <label for="chkCGU" accesskey="U">J'accepte les
conditions générales d'utilisation.</label>
            </li>
            <li>
                <input type="checkbox" id="chkSpreadData"
                       ➥ name="spreadData" value="yes" />
               <label for="chkSpreadData" accesskey="J">Je consens à ce
que mes informations personnelles soient divulguées à des partenaires.
               </label>
            </li>
        </ul>
        <h2>La <span lang="en">newsletter</span> du site</h2>
        <ul>
            <li>
                <input type="radio" id="rbtNoSub" name="newsletter"
                       ➥ value="no" checked="checked" />
               <label for="rbtNoSub" accesskey="R">Je ne désire pas la
recevoir.</label>
            </li>
```

```
            <li>
                <input type="radio" id="rbtWeekly"
                       ➥ name="newsletter" value="weekly" />
                <label for="rbtWeekly" accesskey="B">Je désire la
recevoir hebdomadairement.</label>
            </li>
            <li>
                <input type="radio" id="rbtMonthly"
                       ➥ name="newsletter" value="monthly" />
                <label for="rbtMonthly" accesskey="L">Je désire la
recevoir mensuellement.</label>
            </li>
        </ul>
    </fieldset>
    <p class="submit"><input type="submit" id="btnSubmit" value="Mettre
à jour mon profil" accesskey="M" title="Alt+M" /></p>
</form>
```

- Pour une case à cocher isolée, on précise la valeur afin d'éviter de dépendre du navigateur pour la valeur par défaut (suivant les cas on obtient `on`, `1`, `yes`...), ce qui compliquerait le test côté serveur. Tester simplement que le champ est présent dans la requête ne serait pas forcément très robuste.
- Remarquez l'attribut bascule `checked` et sa seule valeur autorisée.
- Notez la précision de la langue pour le terme « newsletter », qui permettra notamment aux lecteurs d'écran de le prononcer correctement à l'intention des non-voyants ou malvoyants. La CSS pourrait même basculer en italique de tels éléments, par exemple (sélecteurs d'attribut CSS 2, ou de langue CSS 3).
- Les boutons radio mutuellement exclusifs portent le même nom, mais des valeurs différentes. Le champ n'est envoyé qu'une fois, pour le bouton sélectionné.
- Le bouton d'envoi, champ de type `submit`, n'a pas de libellé externe : il précise donc son `accesskey` et la rend consultable avec `title`. Il est par ailleurs inutile de lui donner un attribut `name`, ce qui enverrait un champ au serveur alors qu'il n'y a pas plusieurs modes d'envoi parmi lesquels choisir...

Le fichier HTML complet pèse 5 Ko, et la CSS en ajoute 1,5, soit un total de 6,5 Ko. Croyez-moi sur parole, l'équivalent en HTML des années 1990 est beaucoup plus lourd, alors qu'il est aussi beaucoup plus rigide et incapable de s'adapter d'un mode de consultation à un autre.

La figure A–1 vous montre le résultat sans aucune feuille de styles.

Figure A–1
Notre formulaire sans aucune mise en page

Cela fait certes peur, mais ce n'est pas grave : *le contenu de la page a du sens, beaucoup de sens*. La figure A–2 montre le formulaire, avec une feuille de styles et l'application d'un petit script de décoration automatique de libellés, comme celui vu au chapitre 3 dans la section *Décoration automatique de labels* :

Quand je vous disais que les CSS font des miracles...

Il est par ailleurs possible d'y ajouter un script non intrusif de validation automatique de formulaire, comme vu au chapitre 3, dans la section *Validation automatique de formulaires*. Vous trouverez aussi en fin d'annexe, dans la section *Pour aller plus loin...*, l'URL d'un article OpenWeb dédié à la conception de formulaires sémantiques.

Figure A–2
Le même formulaire, mis en forme par CSS et un script complémentaire

Un tableau de données à en-têtes groupés

Prenons maintenant l'exemple d'un tableau de données. La balise `table` et ses collègues : `thead`, `tbody`, `tfoot`, `tr`, `th` et `td`, ne sont pas à éviter comme la peste ! Simplement, elles servent à décrire des tableaux de données, pas à réaliser une mise en page aussi rigide (et lourde) qu'un parpaing.

Supposons que nous souhaitions réaliser un tableau représentant en abscisse des produits (classés par catégories) et en ordonnée des modes de livraison, eux-mêmes déclinés par zone géographique de livraison. On aimerait que les en-têtes du tableau persistent au-delà d'un saut de page à l'impression. On aimerait aussi qu'il soient sémantiques et un minimum accessibles aux utilisateurs de lecteurs d'écran.

Voici comment procéder...

Listing A-6 Un tableau bien structuré, sémantique et accessible

```
<table id="shipment" summary="Options de livraisons et leurs coûts">
    <caption>Coûts de livraison par mode et par zone géographique
    </caption>
    <thead>
        <tr>
            <th rowspan="2" class="bodyHeading">Mode</th>
            <th colspan="3">T-shirts</th>
            <th colspan="4">Gadgets</th>
        </tr>
        <tr>
            <th>1&ndash;10</th>
            <th>11&ndash;50</th>
            <th>&gt; 50</th>
            <th>&lt; 1kg</th>
            <th>1&ndash;5kg</th>
            <th>5&ndash;10kg</th>
            <th>&gt; 10kg</th>
        </tr>
    </thead>
    <tbody>
        <tr><th colspan="8" class="zone">Europe</th></tr>
        <tr>
            <th>ColiPoste</th>
            <td>2€</td>
            <td>5€</td>
            <td>15€</td>
            <td>5€</td>
            <td>15€</td>
            <td>22€</td>
            <td>30€</td>
        </tr>
        <tr>
            <th>Colissimo</th>
…
        </tr>
    </tbody>
    <tbody>
        <tr><th colspan="8" class="zone">Amérique du Nord, Maghreb,
Australie</th></tr>
…
    </tbody>
</table>
```

- Notez l’attribut `summary`, qui résume le contenu de la table, et l’élément fils `caption`, qui lui fournit un titre explicite.

- Les en-têtes sont, logiquement, dans `thead`. Les cellules qui y figurent ne fournissent que des titres, ce sont donc des `th`.
- Les différents corps de données (ici un par zone géographique) sont des `tbody`. Chaque corps a ici un sous-titre dédié, réalisé avec une première ligne ne comportant qu'un `th` sur toute la largeur.
- Chaque ligne démarre par un titre, donc un `th`, puis n'a que des données, donc des `td`.

Le fichier HTML pèse 2 Ko, la CSS, quoique bien aérée, ne pèse que 610 octets. Voici l'aspect du tableau, doté de cette feuille de styles minimaliste :

Figure A–3
Notre tableau avec un style léger

COÛTS DE LIVRAISON PAR MODE ET PAR ZONE GÉOGRAPHIQUE

Mode	T-shirts			Gadgets			
	1-10	11-50	> 50	< 1kg	1-5kg	5-10kg	> 10kg
Europe							
ColiPoste	2€	5€	15€	5€	15€	22€	30€
Colissimo	5€	10€	30€	12€	27€	36€	45€
Éclair	15€	20€	45€	20€	50€	70€	100€
Amérique du Nord, Maghreb, Australie							
ColiPoste	4€	10€	30€	10€	30€	44€	60€
Colissimo	10€	20€	60€	24€	54€	72€	90€
Éclair	30€	40€	90€	40€	100€	140€	200€

Si on souhaite rendre maximale l'accessibilité, en particulier à destination des lecteurs d'écran (non-voyants, malvoyants) et des personnes souffrant d'un handicap cognitif, les éléments `th` et `td` disposent d'attributs `abbr`, `headers` et `scope` extrêmement utiles. Vous pourrez en apprendre davantage sur cet aspect et les tableaux en général ici : http://pompage.net/pompe/autableau/.

Un didacticiel technique

Pour finir, penchons-nous sur les balises sémantiques en ligne, c'est-à-dire conçues pour indiquer le sens d'un fragment de texte. Nous allons prendre l'exemple d'une documentation technique qui, après avoir référencé quelques sources et cité un fragment pertinent de l'une d'elles, fournit aux lecteurs un morceau de code source et les guide à travers quelques manipulations clavier dans leur console.

Listing A-7 Une documentation au balisage hautement sémantique

```
<h1>Lister les libellés d'un document</h1>

<p>Nous allons apprendre à lister les libellés d'un document HTML à l'aide
du DOM niveau 2 (noyau et HTML). On se reposera essentiellement sur
<code>document.getElementsByTagName</code>. Cette méthode, je cite,
<q cite="http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-A6C9094"
lang="en">Returns a <code>NodeList</code> of all the <code>Element</code>s
with a given tag name in the order in which they are encountered in a
preorder traversal of the <code>Document</code> tree.</q>.</p>

<p>Qu'est-ce qu'une <code>NodeList</code> ? C'est une interface
d'énumération de n&oelig;uds. La spécification la décrit ainsi :
</p>

<blockquote cite="http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-
536297177" lang="en">
    <p>The <code>NodeList</code> interface provides the abstraction of
an ordered collection of nodes, without defining or constraining how
this collection is implemented. <code>NodeList</code> objects in the DOM
are live.</p>
    <p>The items in the <code>NodeList</code> are accessible via an
integral index, starting from 0.</p>
</blockquote>

<p>Voici le code nécessaire. Par simplicité, on suppose qu'une fonction
<code>getInnerText</code> existe :</p>

<pre class="code">
<b>function</b> printLabelsInfo() {
    <b>var</b> msg = '';
    <b>var</b> labels = document.getElementsByTagName('label');
    <b>for</b> (<b>var</b> index = 0; index &lt; labels.length; ++index) {
        <b>var</b> label = labels[index];
        msg += getInnerText(label);
        <b>if</b> (label.hasAttribute('for') &amp;&amp;
                document.getElementById(label.htmlFor))
            msg += ' -&gt;' + label.htmlFor;
        <b>if</b> (label.hasAttribute('accesskey'))
            msg += ' (Alt+' + label.accessKey)';
        msg += '\n';
    }
    alert(msg);
} <i>// printLabelsInfo</i>
</pre>

<p>Sauvegardez ce code dans un fichier <tt>labels.js</tt>. Vous
remarquez qu'il est de faible taille :</p>
```

```
<pre class="console">
<samp>$</samp> <kbd>ls -l</kbd>
<samp>total 4
-rw-r--r-- 1 demo demo 530 2006-08-23 16:58 labels.js
$</samp>
</pre>
```

- On voit ici des exemples de nombreuses balises sémantiques. On le voit, `code` représente du code de façon générale (lorsqu'il s'agit d'un nom de variable dans un code source, on utilise de préférence `var`), et `tt` sert à indiquer les textes perçus comme du contenu informatique mais qui ne sont pas forcément affichés par le système (`samp`), ni saisis (`kbd`). Classiquement, les noms de fichiers.
- Dans un code source, on peut vouloir améliorer la mise en page, par exemple en mettant les mots réservés en gras et les commentaires en italique, et peut-être en gris. Ce n'est que de la coloration syntaxique, sans sémantique : `i` et `b` sont parfaits pour cela !
- Remarquez les citations brèves, en ligne, avec `q` (et son attribut `cite` pour avoir la source), et celles plus longues, en bloc, avec `blockquote` (et son attribut `cite`, là encore).

Là aussi, le HTML et la CSS ne totalisent que 3 Ko. Voici le résultat, doté d'une légère feuille de styles.



Figure A–4
Notre fragment de documentation avec un style léger

Pour aller plus loin...

Livres

HTML avec CSS et XHTML Tête la première
Elisabeth Freeman, Eric Freeman
O'Reilly France, août 2006, 720 pages (de bonheur)
ISBN 2-841-77413-9

Introduction à HTML et CSS
Eric Sarrion
O'Reilly, février 2006, 231 pages
ISBN 2-841-77400-7

HTML et CSS 2
Molly Holzschlag
Pearson Education, septembre 2005, 330 pages
ISBN 2-744-01994-1

Design web : utiliser les standards
Jeffrey Zeldman
Eyrolles, avril 2005, 414 pages (d'évangile)
ISBN 2-212-11548-2

HTML : précis et concis
Jennifer Niederst
O'Reilly, avril 2002, 122 pages (d'aide-mémoire)
ISBN 2-841-77157-1

Sites

- Le W3C évidemment : http://w3.org. Et les recommandations fondamentales (seule la version anglaise a valeur de référence) :
  - HTML 4.01 : http://w3.org/TR/html401/ (version française à cette adresse : http://www.la-grange.net/w3c/html4.01/cover.html)
  - XHTML 1.0 : http://w3.org/TR/xhtml1/ (version française à cette adresse : http://www.la-grange.net/w3c/xhtml1/)
- OpenWeb, un excellent site didactique en français, joli et pratique : http://openweb.eu.org/. On notera en particulier :
  - http://openweb.eu.org/articles/xhtml_une_heure/
  - http://openweb.eu.org/articles/respecter_semantique/

  - http://openweb.eu.org/articles/html_au_xhtml/
  - http://openweb.eu.org/articles/formulaire_accessible/
- Pompage, excellent site voué à la traduction francophone d'articles de grande qualité sur la conception web : http://pompage.net. On notera particulièrement, histoire de se mettre en appétit, les articles suivants :
  - http://pompage.net/pompe/bonpiedstandards/
  - http://pompage.net/pompe/cederholm/
  - http://pompage.net/pompe/separation/
  - http://pompage.net/pompe/autableau/
  - http://pompage.net/pompe/listes/
  - http://pompage.net/pompe/listesdefinitions/
  - http://pompage.net/pompe/doctypecontenttype/
- AccessiWeb, la cellule accessibilité de BrailleNet, regorge de documentations et manuels pratiques en français pour rendre vos sites et créations numériques plus accessibles : http://www.accessiweb.org/.
- Opquast est un référentiel français de bonnes pratiques de conception web classées suivant de multiples axes, doté en plus d'un outil d'aide à l'estimation et au suivi de la qualité de vos sites : http://www.opquast.com/.
- Alsa Créations est un site didactique regorgeant d'excellents articles sur les standards, XHTML et CSS : http://www.alsacreations.com/. Un forum permet par ailleurs aux visiteurs de s'entraider.
- A List Apart est un incontournable : http://alistapart.com/.

B

Aspect irréprochable et flexible : CSS 2.1

Cette annexe n'a pas l'intention de vous apprendre CSS. Le sujet est si vaste, et les subtilités si nombreuses, qu'il faut pour cela des livres entiers (et il y en a d'excellents, voir la fin de l'annexe). Il s'agit juste de faire le point sur la prise en charge des CSS dans les navigateurs, de rappeler quelques grandes notions fondamentales (structure des règles, cascade, modèles fondateurs), et de fournir un tour d'horizon des sélecteurs et propriétés.

Notez que ce survol sera moins détaillé que pour l'annexe A, dont l'objectif était d'attirer votre attention sur des bons ou mauvais usages fréquents et de mettre en avant des attributs sous-employés, ce qui exigeait de longues descriptions et quelques exemples solides. Ici, on a plutôt une sorte de référence laconique.

Statut, état et vocabulaire

CSS est un standard W3C, publié sous forme de recommandations. À l'exception de (X)HTML, il n'y a sans doute pas de standard du Web plus utilisé que CSS.

Les versions de CSS

Tableau B–1 Les versions du standard CSS

Version	Description
1.0	Première édition le 17 décembre 1996 (oui, les CSS ont dix ans !), avec une révision le 11 janvier 1999. Bien qu'assez simple, cette première mouture fournissait déjà l'essentiel des modèles des boîtes et de mise en forme visuelle.
2.0	Sortie le 12 mai 1998, cette version ajoutait énormément de choses : les types de média, la valeur spéciale `inherit`, la pagination, des fonctionnalités d'internationalisation, une gestion détaillée des tableaux, les positionnements absolu, relatif et fixe, la gestion du débordement et de la troncature de contenu, le contenu généré, et j'en passe.
2.1	Toujours à l'état d'ébauche (cinquième révision le 11 avril 2006), cette version constitue surtout un affinage et une série de corrections et de précisions à la version 2.0. Elle constitue l'état « actuel » de CSS dans la plupart des navigateurs.
3.0	Dans la tendance actuelle de modularisation au W3C, CSS 3 est constituée de 37 (vous avez bien lu) modules, chacun faisant l'objet d'un travail indépendant. À l'heure où j'écris ces lignes, aucun n'est totalement finalisé ! Huit le sont presque (CR, *Candidate Recommandation*), trois sont au stade précédent (*last call*), dont celui des sélecteurs (extrêmement attendu !) et celui des polices de caractères, et le reste est en ébauche (parfois figée depuis des années), voire même pas démarré (cinq modules). En somme, le travail n'avance pas.

Prise en charge actuelle

Comme on peut le voir dans l'avant-propos, la prise en charge de CSS est généralement bonne, avec quelques écarts. Pour simplifier, on peut dire qu'avec la sortie de MSIE 7, tous les navigateurs véritablement répandus prendront décemment en charge CSS 2.1.

Voici tout de même un petit bilan (mais n'hésitez pas à consulter les informations techniques de vos navigateurs cibles : ces données évoluent souvent). Attention : les informations pour MSIE supposent une page en mode strict (utilisation d'un `DOCTYPE` sur une DTD (X)HTML `Strict`). Voir l'annexe A à ce sujet.

Les adjectifs expriment le degré de prise en charge.

Navigateur	CSS 1	CSS 2	CSS 2.1	CSS 3
MSIE 6	Bon	Moyen	-	-
MSIE 7	-	-	Très bon	Bon
Mozilla, Firefox, Camino	-	-	Très bon	Léger
Safari	-	-	Très bon	-
Opera	-	-	Excellent	Bon
Konqueror (KDE 3.5)	-	-	Très bon	Léger

Les parties les moins bien prises en charge de CSS 2.1 concernent surtout les feuilles de styles auditives (*aural style sheets*), qui visent à contrôler la lecture vocale des pages : sexe de la voix, richesse, prosodie, intonation, débit... Ce ne sont pas vraiment les navigateurs qui doivent implémenter cela, mais les lecteurs d'écran !

Le jargon CSS

Lorsqu'on parle de CSS, certains termes précis reviennent régulièrement. Reprenons-les ici avant de poursuivre la lecture de cette annexe.

- Une **propriété** est un nom spécifique représentant un aspect CSS. Par exemple, `font-weight` gère l'épaisseur des caractères (normale, grasse, etc.) tandis que `white-space` régit le traitement des espacements et retours chariot. Chaque propriété dispose d'une série de valeurs possibles, avec généralement plusieurs syntaxes autorisées. Elle peut avoir une valeur par défaut, bénéficier ou non de l'héritage par cascade, et ne s'appliquer qu'à certains éléments.
- Un **sélecteur** est un texte respectant une syntaxe spéciale qui permet de désigner des éléments de la page en fonction de critères variés : leur nom de balise, leur ID, leurs classes CSS, la présence d'un attribut, leur position dans le document, la langue de leur contenu, etc. En combinant des sélecteurs, on peut décrire une extraction très fine et précise d'éléments dans le document.
- Une **règle** est l'élément de base d'une feuille de styles CSS : elle définit une série de propriétés pour les éléments désignés par un ou plusieurs sélecteurs.

Prêt pour un exemple qui n'a d'autre intérêt que justement de servir d'exemple ? C'est parti. Dans le code CSS suivant :

```
#examples tr.critical>th:first-child, th.critical {
    background: red;
    color: white;
    font-weight: bold;
}
```

- `#examples` est un sélecteur d'ID, l'espace qui suit est un sélecteur de descendant, `tr` est un sélecteur de type, `.critical` est un sélecteur de classe, le chevron fermant (>) est un sélecteur d'élément fils, `th` est un sélecteur de type, `:first-child` est un pseudo-élément et la virgule est un opérateur de groupement entre deux combinaisons de sélecteurs. Si vous êtes perdu, ne vous en faites pas, on reverra ces sélecteurs plus loin.
- `background`, `color` et `font-weight` sont des propriétés, dont `red`, `white` et `bold` sont les valeurs respectives.
- L'ensemble du fragment constitue une règle.

Histoire de ne pas vous faire languir si vous n'êtes pas très au point côté CSS, la règle s'applique ici aux éléments suivants :

- Les éléments `th`, apparaissant comme premier élément fils dans les `tr` ayant une classe CSS `critical`, lesquels apparaissent quelque part dans un élément d'ID `examples`.
- Les éléments `th` ayant une classe CSS `critical`.

Bien comprendre la « cascade » et l'héritage

Les feuilles de styles applicables à un document peuvent avoir trois origines :

- **Le navigateur**, qui fournit généralement une feuille de styles par défaut (grâce à laquelle les titres sont plus gros, en italique, etc.).
- **L'auteur** du document ; la feuille est alors généralement un fichier `.css` associé par une balise `link`.
- **L'internaute**, qui peut appliquer une feuille de styles utilisateur (pour peu que son navigateur l'autorise).

La cascade détermine comment sélectionner, propriété par propriété, la valeur spécifiée à retenir pour un élément donné. Contrairement à une idée répandue, elle ne décrit pas le mécanisme d'héritage, bien plus simple et qui n'a pas besoin d'elle. Je reviendrai sur cet héritage un peu plus tard.

Le sens de la cascade

La cascade se déroule approximativement de la façon suivante :

1. On récupère toutes les règles applicables (en vertu de leurs sélecteurs) à l'élément concerné, pour le média courant (saviez-vous qu'on peut restreindre une feuille de styles à certains médias seulement, comme l'écran, l'imprimante ou la vidéoprojection ?).

2 On les trie par importance (présence de la clause `!important`) et par origine, dans l'ordre suivant (du moins prioritaire au plus prioritaire) :
   a. règles « par défaut » venant du navigateur ;
   b. règles normales de l'internaute ;
   c. règles normales de l'auteur ;
   d. règles importantes de l'auteur ;
   e. règles importantes de l'internaute ;
3 À priorité égale, on les trie par **spécificité** (voir section suivante).
4 Enfin, on trie par ordre d'apparition : quand plusieurs définitions ont la même priorité et la même spécificité, la dernière est retenue.

La syntaxe `!important` en fin de déclaration de propriété est là pour améliorer l'accessibilité en garantissant à l'internaute la possibilité de remplacer une propriété qui nuit à son confort : il suffit de déclarer une autre valeur dans une feuille de styles utilisateur, dotée de l'attribut final `!important`.

Calcul de la spécificité d'une règle

Voyons à présent comment calculer la spécificité, ou le « poids », d'une règle CSS. La spécificité est une valeur à quatre composantes : `a-b-c-d`. Ces composantes étant souvent de valeur inférieure à 10, on a en général un nombre décimal entre 0 et 9999. C'est le poids. Mais comment déterminer la valeur de chaque composante ?

a. La composante `a` vaut `1` pour un attribut HTML `style`, `0` pour une feuille externe.

b. La composante `b` est le nombre de sélecteurs d'ID impliqués (`#truc`).

c. La composante `c` est le nombre de sélecteurs sur attributs (`[truc]`, `.bidule`, `:lang(chose)`), ainsi que les pseudoclasses (par exemple `:hover` ou `:focus`).

d. La composante `d` est le nombre de sélecteurs de type (par exemple `h1`) et de pseudo-éléments (par exemple `:first-child`).

Vous avez besoin d'exemples ? D'accord !

Tableau B–2 Exemples de calculs de spécificité

Sélecteurs	a	b	c	d	Total
*	0	0	0	0	0
`li`	0	0	0	`li` →1	1
`li:first-line`	0	0	0	`li`, `:first-line` →2	2
`ul li`	0	0	0	`ul`, `li` →2	2
`ul ol + li`	0	0	0	`ul`, `ol`, `li` →3	3

Tableau B–2 Exemples de calculs de spécificité (suite)

Sélecteurs	a	b	c	d	Total
`h1 + *[rel=up]`	0	0	`[rel=up]` →1	`h1` →1	11
`ul ol li.red`	0	0	`.red` →1	`ul`, `ol`, `li` →3	13
`li.red.level`	0	0	`.red`, `.level` →2	`li` →1	21
`#x34y`	0	`#x34y` →1	0	0	100
`style="..."`	1	0	0	0	1000

Que se passe-t-il avec la présentation dans HTML ?

Le détail obscur : si, horreur ultime, le HTML contenait des attributs de présentation (vous savez, ces momies que sont `bgcolor`, `border`, `align`, etc.), le navigateur peut les prendre en compte à condition de les considérer comme des règles placées en début de feuille de styles auteur, avec une spécificité zéro. En d'autres termes, elles seront prioritaires uniquement sur le style par défaut fourni par le navigateur et céderont devant toute spécification de type CSS.

L'héritage

L'héritage est un mécanisme par lequel des éléments voient certaines de leurs propriétés « hériter » leur valeur de celle utilisée par un élément conteneur.

Par exemple, si `body` précise une `font-size` de valeur `large`, les éléments `p` à l'intérieur de `body` (directement ou non) utiliseront la même valeur pour leur propre `font-size`, sauf instruction contraire.

Pour bien comprendre le rôle de l'héritage, il faut examiner comment le navigateur détermine la valeur concrète d'une propriété. Il ne suffit pas de prendre la valeur spécifiée dans la CSS, loin de là !

Pour commencer, sachez qu'un navigateur doit avoir défini une valeur concrète pour toutes les propriétés de tous les éléments lorsqu'il a achevé le *rendering* de la page ! Cela fait donc beaucoup de valeurs, même si on exclut les propriétés qui ne s'appliquent pas au média courant (par exemple, les propriétés de pagination lorsqu'on affiche à l'écran).

Cette valeur concrète (*actual value* dans la spécification) est le résultat d'un calcul en quatre étapes. Dit comme cela, ça semble très compliqué, mais vous allez voir, c'est en fait plutôt naturel.

On commence par déterminer la **valeur spécifiée**, celle qui fait l'objet d'une déclaration ou d'une valeur par défaut. Pour cela, on cherche une valeur indiquée explicitement dans les CSS, en résolvant les conflits éventuels à l'aide de l'algorithme de cascade décrit ci-dessus.

C'est là que l'héritage peut entrer en jeu. Chaque propriété peut ou non en bénéficier automatiquement : cela est précisé dans la spécification CSS 2.1, à l'aide de la caractéristique *Inheritable*. Si on n'a pas trouvé de valeur explicite pour une propriété « héritable », on utilise alors la valeur calculée de l'élément parent (pour lequel toutes les propriétés ont déjà été résolues). On peut obtenir le même effet pour une propriété sans héritage automatique, en définissant la propriété avec la valeur spéciale `inherit`.

De la valeur spécifiée à la valeur concrète

En revanche, s'il n'y a pas d'héritage (la propriété n'est pas définie et elle ne bénéficie pas d'un héritage automatique), on utilise alors la valeur initiale, indiquée dans la spécification CSS 2.1 pour cette propriété.

Deuxième étape : obtenir la **valeur calculée**. Il s'agit d'ajustements à la valeur spécifiée, par exemple la transformation d'URI relatifs en URI absolus, la conversion de tailles en unités relatives (`em`, `ex`...) en unités absolues (`px`), et tout autre changement qui n'a pas besoin d'un véritable *rendering* pour être déterminé. Cette valeur existe pout tout élément, même si la propriété ne s'applique pas à l'élément lui-même, afin de pouvoir utiliser l'héritage sur ses éléments descendants.

Troisième étape : passer à la **valeur utilisée**. On effectue alors les conversions résultant d'un *rendering*, comme transformer une largeur exprimée en pourcentage en une largeur absolue (puisqu'on connaît alors la largeur réelle de son conteneur).

Ultime étape, qui n'intéresse d'ailleurs pas toujours le développeur web : la **valeur concrète**, celle réellement employée. C'est le résultat des contraintes externes (notamment celles du périphérique) sur la valeur utilisée. Par exemple, une largeur absolue peut encore être de 2,8 pixels, alors qu'à l'écran, on ne peut afficher que des pixels entiers ; on arrondira donc à 3. Une police pourrait demander du 13 points, mais si le système de gestion des polices ne peut obtenir cette taille exacte, on ajustera sur la taille la plus proche, par exemple 12 points. Le document est en couleurs, mais l'imprimante n'autorise que les nuances de gris ; on interpole donc les valeurs vers de telles nuances. Vous voyez l'idée.

Voilà certainement plus d'informations que vous ne souhaitiez en avoir. Il est vrai que ce sont surtout les deux premières étapes qui nous intéressent !

Les modèles des boîtes et de mise en forme visuelle

Je vous l'annonce sans ambages : nous n'irons pas dans tous les détails, loin s'en faut. Le modèle des boîtes (*box model*) a fait couler tant d'encre et d'octets qu'on pourrait remplir un volume encyclopédique avec la somme des articles, chapitres et billets sur le sujet.

Je vais simplement couvrir la partie émergée, l'essentiel, le courant, sans me soucier des cas particuliers.

Les côtés d'une boîte : ordre et syntaxes courtes

De très nombreuses propriétés CSS peuvent s'appliquer sur les quatre côtés d'une boîte, ou sur ses côtés horizontaux (haut et bas) et verticaux (droite et gauche), ou sur ses quatre côtés individuellement. Vous trouverez alors toujours des propriétés dites « abrégées » (*shorthand properties*), permettant de spécifier ces côtés d'un seul coup.

Les variantes et l'ordre sont toujours les mêmes, aussi je les précise ici une bonne fois pour toutes. Prenons l'exemple de la propriété `margin`, qui régit la marge externe d'une boîte, comme nous le verrons à la prochaine section.

Il existe des propriétés dédiées pour chaque côté : `margin-top`, `margin-right`, `margin-bottom` et `margin-left`. Il existe aussi la propriété courte `margin`, justement, qui peut prendre les trois formes suivantes.

Tableau B–3 Variantes de définition d'une propriété courte

Variante de format	Résultat
`margin: 1em;`	Même marge pour les quatre côtés : `1em`.
`margin: 1em auto;`	Marge de `1em` en haut et en bas, marge automatique (centrage de bloc) à droite et à gauche.
`margin: 1em 0 0.5em`	Marge haute de `1em`, droite et gauche de `0`, basse de `0.5em`
`margin: 0.5em 0 1em 2ex;`	Marge haute de `0.5em`, droite de `0`, basse de `1em`, gauche de `2ex`.

Si vous avez du mal à retenir l'ordre, pensez à ceci : on part du haut et on suit les aiguilles d'une montre.

Unités absolues et relatives

CSS fournit de nombreuses unités pour exprimer des tailles (de police, de bloc...). Certaines sont absolues et d'autres relatives. Certaines n'ont de sens que pour certains médias. En voici un récapitulatif.

Tableau B–4 Unités de CSS 2.1

Unité	Nature	Signification
%	relative	Pourcentage de la valeur calculée héritée (ou initiale, pour body). Par exemple dans un body avec `font-size: 1.5em`, un p avec `font-size : 150 %` aura en fait un `font-size` de `2.25em`.

Tableau B–4 Unités de CSS 2.1 (suite)

Unité	Nature	Signification
`cm`	absolue	Centimètres. S'ajuste normalement aux résolutions de l'imprimante comme de l'écran (72 ou 96 ppp).
`em`	relative	La hauteur de la police courante, définie comme le *em square* typographique. En gros, en dépit de ses origines liées à la largeur, c'est à peu près la hauteur d'un caractère majuscule. Unité de choix pour les marges et espacements en général (même si j'ai plus tendance à limiter aux côtés haut et bas) et aux hauteurs de blocs.
`ex`	relative	La hauteur d'un caractère minuscule sans jambe (on utilise généralement « x »). Correspond souvent à la largeur moyenne d'un caractère : un conteneur de largeur `10ex` peut contenir environ 10 caractères. Unité pertinente aussi pour les marges et espacements en général (je l'utilise surtout pour les bords droit et gauche, et les largeurs de blocs).
`in`	absolue	Pouces (*inches*). Je rappelle qu'un pouce vaut 2,54 cm. Même remarque que pour `cm`.
`mm`	absolue	Millimètres. Même remarque que pour `cm`.
`pc`	absolue imprimante	Picas (unité typographique). Cette merveilleuse unité semi-préhistorique vaut 12 points (voir `pt`).
`pt`	absolue imprimante	Points. Unité typographique qui n'a vraiment de sens que pour une CSS d'impression (média `print`). CSS 2.1 cale le point à 1/72 de pouce, soit environ 0,35 mm. Pas étonnant que beaucoup préfèrent le système métrique aux unités britanniques. Notez que lors de l'impression, caler les tailles de fontes en points est bien plus pertinent qu'en `em`/`ex`, car cela favorise l'homogénéité à travers les imprimantes et systèmes.
`px`	classée relative, mais plutôt « absolue écran »	Pixels. Unité un peu fourbe, car selon la résolution de l'écran (pas 1024 × 768, mais 72 ppp), un pixel sera plus ou moins gros. Sur Mac OS (avant OS X) notamment, les pixels étaient notoirement plus petits, ce qui rendait toute police `10 px` illisible. N'utilisez cette unité que pour des éléments dont la taille doit coller à des ressources non redimensionnables, comme des images. Pour du texte, je suis partisan de la proscrire, surtout sur des petites et moyennes tailles (moins de `30 px`), car le texte ne peut alors pas être redimensionné (notamment agrandi) suivant les besoins de l'internaute.

Je vous recommande très, très chaudement d'utiliser des **unités relatives** pour tout ce qui a vocation à suivre la taille du texte. Cela inclut souvent les bordures, les marges, les positionnements, et suivant l'esthétique retenue, parfois les espacements (*paddings*).

Ainsi, l'ensemble de votre page s'ajustera mieux lorsque l'internaute modifiera la taille de base des polices de caractères pour y voir plus clair, ce que tous les navigateurs répandus permettent de faire.

On donnera, notamment, priorité à `em`, `ex` et `%` à l'écran, et `pt` à l'impression !

Dernière remarque : pour une taille de zéro, on ne précise pas l'unité en général. C'est en effet mal vu par les esthètes (on parle de *bad form*), car zéro est zéro, quelle que soit l'unité.

Marge, bordure et espacement

Une boîte est dotée, de l'extérieur vers l'intérieur, d'une marge (espacement externe à sa bordure), d'une bordure (avec une couleur, un type de trait et une épaisseur), d'un espacement (*padding* en anglais, espacement interne à la bordure), et tout au centre, de son contenu. Le schéma suivant illustre cette imbrication :

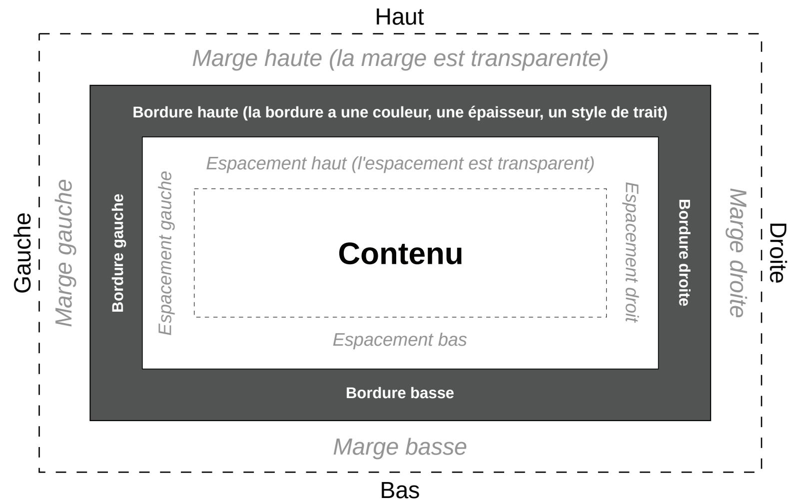


Figure 2–1
Imbrication de marge, bordure, espacement et contenu

À présent, un point très important, car contre-intuitif : la largeur et la hauteur d'un élément sont en réalité les dimensions de son contenu. Prenons la règle suivante :

```
div#demo {
    width: 40ex;
    padding: 1ex;
    border: 1ex solid gray;
    margin: 1em 1ex;
}
```

Le `div` d'ID `demo` aura une « largeur visuelle » de 44ex ! En effet :

$$40ex + 2 \times 1ex + 2 \times 1ex = 44ex$$

Il s'agit de sa largeur de contenu, à laquelle on ajoute les largeurs des espacements gauche et droit ainsi que celles des bordures gauche et droite.

Éléments en ligne et de type bloc

La plupart des éléments censés faire partie du flux d'un texte sont dits « en ligne » (*inline*). Ceux censés apparaître comme des blocs hors du flux du texte sont dits « de type bloc » (*block-level*).

Il existe en réalité toute une gamme de cas spécialisés, définis par la propriété `display`. CSS 2 a notamment ajouté toute une série de valeurs spécifiques aux composants de tableaux de données.

Des éléments naturellement en ligne (par exemple `span`, `em`, `strong`, `code`) peuvent devenir de type bloc à l'aide de la propriété `display`, et réciproquement.

Je mentionne cela parce qu'un élément en ligne a certaines limitations en terme de modèle des boîtes. Par exemple, il n'a pas de marge verticale (haute ou basse).

Éléments remplacés

Voici l'astuce du jour... Si vous lisez la spécification CSS 2.1, vous trouverez un peu partout le terme « élément en ligne non remplacé » (*non-replaced inline element*). On peut légitimement se demander à quoi rime ce « non remplacé », qu'on voit un peu partout. La réponse se cache à la section 3.1 de la recommandation, dont le titre, « Conformité », ne nous l'aurait pas laissé supposer.

Un élément remplacé est un élément hors du champ d'action du formateur CSS, en tout cas en ce qui concerne ses dimensions propres. Dans la pratique, c'est l'image obtenue par chargement d'une balise `img` ou alors le contenu chargé par une balise `object`. Tout autre élément en ligne est donc non remplacé, ce qui rend cette précision superflue dans la plupart des cas.

Fusion des marges

Les marges expriment généralement un besoin d'espace autour d'un élément. Cependant, dans le cas des marges verticales, il est souvent inutile de cumuler les marges. Imaginons qu'on utilise la règle suivante :

```
p { margin: 1em 0; }
```

Cette règle demande des marges haute et basse, dites « marges verticales », de `1em` (hauteur moyenne d'un texte dans la police de caractères active), et pas de marges horizontales (droite et gauche).

Pourtant, lorsqu'on va enchaîner deux paragraphes, il serait curieux, voire agaçant, que ces marges se cumulent, produisant un espacement de 2 lignes de haut entre les paragraphes. Ce n'est généralement pas l'intention de l'auteur de la règle. Pour cette raison, dans de nombreuses situations, CSS 2.1 fusionne les marges verticales. Pour faire simple, lorsqu'on a deux marges verticales adjacentes, on n'utilise que la plus grande des deux.

Attention toutefois, ce mécanisme n'a pas court dans tous les cas. La section 8.3.1 de la recommandation W3C détaille les cas concernés, dont la liste exhaustive va au-delà du rôle de cette annexe.

Le modèle W3C et le modèle Microsoft

Voici la source de bien des soucis, en tout cas il n'y a pas si longtemps (de nos jours, à peu près tous les professionnels ont pris le pli). Les développeurs de MSIE avaient à l'origine mal interprété le sens des propriétés `width` et `height`. Ils pensaient que ces dimensions incluaient l'espacement et la bordure.

Ce n'est pas déraisonnable : dans la pratique, quand vous faites référence aux dimensions d'une boîte, vous parlez des bords extérieurs de la boîte, pas de la taille du contenu calé dans la boîte par des protections en mousse ou en polystyrène. Hélas, ce n'est pas le sens retenu par la recommandation W3C.

Le souci est apparu dès que les autres navigateurs ont rattrapé MSIE 5 sur le terrain des CSS. Ces navigateurs implémentaient correctement la recommandation, et on s'est retrouvé avec des incohérences flagrantes entre l'affichage MSIE et celui des autres navigateurs. Je m'en souviens encore, et croyez-moi, ça nous a tous fait fulminer.

MSIE 6 a donc sorti une astuce, le *doctype switching*, expliquée au début de l'annexe A, qui a rapidement été reprise par les autres navigateurs pour laisser l'auteur de la page décider de l'interprétation de `width` et `height`.

Lorsqu'une page déclare une DTD stricte (HTML 4.01 Strict ou XHTML 1.0 Strict, par exemple), le navigateur fonctionne en *Strict Mode*. On respecte alors la définition du W3C. Si une page ne déclare pas de DTD ou déclare une DTD non stricte, le navigateur utilise le *Quirks Mode*, qui simule le comportement de l'ancien MSIE, en considérant que l'espacement et la bordure sont compris dans la taille.

En réalité, de nombreux navigateurs utilisent aussi le *Quirks Mode* pour émuler d'autres erreurs CSS de MSIE, et pas seulement de la version 5.5... Mais c'est un autre débat.

En conclusion : **déclarez toujours une DTD, et qu'elle soit stricte, bon sang !**

Tour d'horizon des sélecteurs

CSS 2.1 définit de nombreux sélecteurs (et si vous voyiez CSS 3 !). En voici un tour d'horizon. Attention, certains ne sont pas pris en charge par MSIE 6, voire par quelques autres navigateurs.

Tableau B–5 Les sélecteurs de CSS 2.1

Sélecteur	Description
`*`	Sélecteur universel. Correspond à tous les éléments.
`balise`	Sélecteur de type. Correspond aux éléments ayant ce nom de balise.
`balise1 balise2`	Sélecteur de descendants. Correspond aux éléments `balise2` quelque part à l'intérieur d'un élément `balise1`, à quelque niveau de profondeur que ce soit.
`.classe`	Sélecteur de classe. En (X)HTML, équivalent à `*[class~="classe"]` (voir plus bas).
`#unID`	Sélecteur d'ID. Correspond à l'élément dont l'attribut `id` à la valeur `unID`.
`balise:link` `balise:visited`	Pseudoclasses de lien. Correspond aux éléments `balise` (qui doivent être des sources de lien, donc généralement des éléments a) dont la cible n'a pas encore (ou a déjà, respectivement) été visitée.

`balise:hover`
`balise:active`
`balise:focus`
	Pseudoclasses dynamiques. Correspond aux éléments `balise` à certains moments de l'interaction utilisateur. Respectivement, il s'agit d'un survol souris, d'une activation (clic ou touche `Entrée`, par exemple) ou de l'obtention du focus (arrivée sur l'élément par tabulation, par exemple).

`balise1 > balise2`	Sélecteur de fils. Correspond aux éléments `balise2` dont l'élément père est un élément `balise1`. Plus restrictif, donc que l'espace, qui n'a pas de limite de profondeur.
`balise1 + balise2`	Sélecteur d'élément adjacent. Correspond à un élément `balise2` dont le précédent élément frère est de type `balise1`. Par exemple, `h2 + p` sélectionne les paragraphes qui suivent immédiatement des titres de deuxième niveau.
`balise:first-child`	Pseudoclasse de premier fils. Correspond à tout élément `balise` étant le premier élément fils de son élément père. Par exemple, permet de sélectionner le premier paragraphe d'une série.
`balise[attr]`	Sélecteur d'attribut. Correspond à tout élément `balise` ayant un attribut `attr` (quelle qu'en soit la valeur, même vide).
`balise[attr="value"]`	Sélecteur d'attribut. Correspond à tout élément `balise` dont l'attribut `attr` a exactement la valeur `value` (sensible à la casse). Les guillemets sont obligatoires.
`balise[attr~="part"]`	Sélecteur d'attribut. Correspond à tout élément `balise` dont l'attribut `attr` contient une série de valeurs séparées par des espaces, et dont l'une est `part`. Voir l'équivalence du sélecteur de classe.
`balise[attr|="prefix"]`	Sélecteur d'attribut. Correspond à tout élément `balise` dont l'attribut `attr` contient une série de valeurs séparées par des tirets (-), et dont la première est `prefix`. Utile pour l'attribut `lang`, par exemple.
`:lang(code)`	Pseudoclasse de langue. Correspond à tout élément dont la langue correspond à `code`.

Groupement

Afin de limiter la duplication (cauchemar de la maintenance), on peut affecter une série de propriétés à plusieurs sélections (combinaisons de sélecteurs) disjointes, en les séparant simplement par des virgules.

```
h1, h2, h3, h4, h5, h6 {
    font-weight: normal;
}
```

Dans cet exemple, tous les titres utiliseront une épaisseur normale au lieu de la graisse par défaut.

Pseudo-éléments

Un pseudo-élément correspond à une portion de texte à l'intérieur d'un élément, ou à un contenu généré dynamiquement. En tout état de cause, il s'agit d'un fragment du document qui ne correspond pas à un élément, mais fait néanmoins l'objet d'une sélection.

Tableau B–6 Les pseudoclasses de CSS 2.1

Pseudo-élément	Description
`E:first-line`	Première ligne d'un corps de texte. E doit être élément de type bloc, `inline-block`, `table-caption` ou `table-cell`. On évitera donc `span`, par exemple.
`E:first-letter`	Première lettre d'un contenu textuel, sauf si cette lettre est précédée d'un autre contenu (image, tableau en ligne...). Toutes les propriétés CSS ne s'appliquent pas (voir section 5.12.2 de la recommandation). Principalement utilisé pour faire des lettrines. S'applique aux mêmes types d'élément que `:first-line`, plus au type `list-item`.
`E:before` `E:after`	Permettent de générer un contenu avant ou après le texte normal de l'élément, à l'aide de la propriété CSS `content`. Très utilisés pour assortir les liens de petits glyphes ou d'informations sur la langue (attribut HTML `hreflang`)... Très pratique aussi pour une feuille de styles dédiée à l'impression (média `print`), qui peut ainsi afficher, à côté du texte des liens, les URL ciblées.

Tour d'horizon des propriétés

Il est temps de faire le tour des propriétés. Pour toutes celles qui ont des versions courtes, j'ai adopté une notation à lignes multiples, avec les possibilités entre crochets, listées dans l'ordre de leur apparition au sein de la propriété courte.

Chaque ligne supplémentaire constitue une composante optionnelle. La propriété `border` est la plus riche, puisqu'elle va d'une seule propriété consolidée (`border`) jusqu'aux propriétés très ciblées (par exemple `border-left-style`).

Je précise aussi une bonne fois pour toutes que toutes les propriétés peuvent valoir la fameuse valeur `inherit`. Je ne le préciserai pas dans les tableaux.

De l'art de réaliser des CSS légères

CSS fournit de nombreuses syntaxes courtes pour simplifier et alléger vos CSS. Nous avons déjà évoqué ce mécanisme dans le cadre des propriétés de marge, bordure et espacement. Allez donc consulter la recommandation pour le détail des autres propriétés courtes, notamment `font`, une vraie merveille !

Par ailleurs, toute propriété de couleur peut avoir les syntaxes suivantes :

- `transparent`, mot réservé, utile entre autres pour préciser une couleur de fond chaque fois qu'on précise une couleur de texte (évitant ainsi les avertissements des validateurs).
- `#`*`rrvvbb`*, où on représente les composantes rouge, verte et bleue par une valeur hexadécimale (entre `00` et `ff`), de 0 à 255.
- `#`*`rvb`*, lorsque les composantes sont des doublons (par exemple `11`, `44` ou `ff`). Ces couleurs sont censées être plus fiables en rendu que les variantes détaillées (`#463`, équivalent de `#446633`, est censée être plus « garantie » que `#426431`, qui lui ressemble à s'y méprendre).
- `rgb(`*`red`*`, `*`green`*`, `*`blue`*`)`, où les composantes sont indiquées en base décimale, de 0 à 255. À mon sens la plus verbeuse, donc à éviter.
- Mot réservé de couleur, par exemple `white` ou `red`. La recommandation définit en section 4.3.6 les 17 noms autorisés et leurs valeurs exactes.

N'oubliez pas qu'une feuille CSS plus courte est aussi plus rapide à charger, et pas forcément plus difficile à lire !

Propriétés du modèle des boîtes : marges, espacements et bordures

Tableau B–7 Propriétés du modèle des boîtes en CSS 2.1

Propriété	Description
`border` *-[top|right|bottom|left]* *-[width|style|color]*	Bordures. L'épaisseur est exprimée en tant que taille ou à l'aide des mots réservés `thin`, `medium` et `thick`. On compte pas moins de 10 styles, les plus courants étant `solid`, `dotted`, `dashed` et `none` (qui diffère de `hidden` pour les cellules de tableaux !). Au fait, MSIE 7 cessera de dessiner les bordures `dotted` d'épaisseur `1px` en `dashed`. Enfin !
`margin` *-[top|right|bottom|left]*	Marges, donc espacements extérieurs.
`padding` *-[top|right|bottom|left]*	Espacements intérieurs.

Propriétés de formatage visuel : positionnement, largeur, hauteur, baseline

Tableau B–8 Propriétés de formatage visuel en CSS 2.1

Propriété	Description
`clear`	Contrôle le flux autour d'un élément flottant. Peut valoir `none`, `left`, `right` ou `both`.
`clip`	Permet de restreindre la portion affichée d'un élément. Par défaut `auto`, donc affiche toute la boîte. Sinon peut définir un rectangle avec `rect(`*top*, *right*, *bottom*, *left*`)`.
`direction`	Indique le sens du texte dans l'élément. Vaut `ltr` (de gauche à droite, par défaut) ou `rtl`.
`display`	Indique le type de boîte de l'élément. Les valeurs les plus courantes sont `inline`, `block` et `none`, mais il y en a 13 autres, dont 10 relatives aux tableaux !
`float`	Rend un élément flottant, c'est-à-dire extrait du flux normal du texte pour aller se caler quelque part contre les bords du conteneur, le flux du texte s'enroulant autour de lui. Suivant le côté du flottement, vaut `left`, `right` ou bien sûr `none`, sa valeur par défaut.
`height, min-height, max-height`	Régissent la hauteur d'un élément, en collaboration avec `overflow`.

Tableau B–8 Propriétés de formatage visuel en CSS 2.1 (suite)

Propriété	Description
`line-height`	Hauteur minimale de ligne, ce qui inclut le texte et un espacement au-dessus et en dessous du texte. Vaut `normal` par défaut (basé sur les caractéristiques de la police de caractères), sinon un nombre ou un pourcentage (multiplicateur de `font-size`), ou encore une taille spécifique (avec des unités, par opposition à un simple nombre).
`overflow`	Gère le dépassement de contenu vis-à-vis des dimensions souhaitées de la boîte. Vaut par défaut `visible` : la boîte s'adapte, notamment en hauteur. Peut aussi valoir `hidden` (le contenu est tronqué), `scroll` (barres de défilement présentes qu'on en ait besoin ou non) ou `auto` (barres de défilement, si besoin).
`position`	Définit le positionnement de l'élément. Vaut `static` par défaut (positionnement défini par le navigateur suivant les contraintes actives), mais peut aussi valoir `absolute`, `relative` ou `fixed` (lequel est enfin pris en charge par MSIE 7).
`top`, `right`, `bottom`, `left`	Pour un élément positionné, fournit les positions de ses extérieurs de marge (absolues ou relatives, suivant `position`).
`unicode-bidi`	Permet à la valeur de direction de fonctionner également sur un élément en ligne.
`vertical-align`	Fournit l'alignement vertical de contenu en ligne (ou du contenu d'une cellule de tableau). Les valeurs ont toutes rapport aux métriques de typographie : le défaut est `baseline`, et on en a 7 autres dont `middle` et `text-top`, plus la possibilité d'une taille ou d'un pourcentage.
`visibility`	Affiche ou masque un élément, tout en conservant l'espace occupé (contrairement à `display`). Outre les valeurs `visible` et `hidden`, une valeur `collapse` a un sens spécial pour les lignes et groupes (de lignes ou de colonnes) des tableaux.
`width, min-width, max-width`	Régissent la largeur d'un élément.
`z-index`	Utilisée uniquement sur les éléments positionnés. Indique leur position « verticale », entre l'œil de l'internaute et le document si vous préférez. Peut valoir `auto` (défaut) ou un numéro. Règle les questions de recouvrement entre éléments se superposant, par exemple lors d'un glisser-déplacer.

Propriétés de contenu généré automatiquement

CSS 2 a introduit la notion de contenu automatique, principalement sur trois axes :

- Du contenu entièrement synthétisé par la CSS, généralement présent devant ou derrière le contenu natif de l'élément.
- Des indices incrémentaux ; principalement pour les listes, mais aussi pour les titres par exemple (enfin des titres numérotés automatiquement, et hiérarchiquement si on le veut !).
- Des symboles ou images destinés aux listes à puces.

Tableau B–9 Propriétés de contenu automatique en CSS 2.1

<table>
<tr><th>Propriété</th><th>Description</th></tr>
<tr><td>content</td><td>Remplace le contenu de l'élément. On l'utilise principalement sur des sélecteurs de pseudo-éléments :before ou :after, pour ajouter plutôt que remplacer. Peut valoir une foule de choses : un texte fixe, une ressource externe dont on fournit l'URI, l'état d'un compteur, la valeur d'un attribut de l'élément (comme hreflang), l'ouverture ou la fermeture des guillemets ou encore le contrôle du niveau d'imbrication de ceux-ci.</td></tr>
<tr><td>counter-increment</td><td>Incrémente un ou plusieurs compteurs déjà définis. Utilise des paires nom × incrément, l'incrément étant optionnel et valant par défaut 1 (un). Voir l'exemple très parlant de la section 12.4 de la recommandation.</td></tr>
<tr><td>counter-reset</td><td>Définit ou réinitialise un ou plusieurs compteurs, en précisant éventuellement leurs nouvelles valeurs. Même syntaxe que counter-increment.</td></tr>
<tr><td>quotes</td><td>Définit les paires de guillemets à utiliser dans l'élément, niveau par niveau (pour des utilisations imbriquées). La valeur par défaut dépend du navigateur. none supprime tout guillemet (pas très utile...). Sinon, on précise des paires de textes, par exemple '«\00a0' '\00a0»' '"' '"' pour le français (la séquence \00a0 représente une espace insécable). À utiliser en combinaison avec les valeurs open-quote et close-quote de content.</td></tr>
<tr><td>list-style
-[type|position|image]</td><td>Régit l'apparence d'une liste. Le type peut avoir pas moins de 15 valeurs dont 11 de numérotation (listes ordonnées), 3 de puces, et none pour retirer les puces ou numéros. La position vaut inside ou outside (défaut), indiquant si les puces ou numéros s'affichent à l'intérieur ou à l'extérieur de la boîte des contenus. Enfin, image permet de remplacer les puces classiques par une image quelconque, dont on fournit l'URI.</td></tr>
</table>

Propriétés de pagination

Ces propriétés ne s'appliquent que dans le cadre d'un *rendering* sur média paginé, ce qui revient à dire : à l'impression (média print). On dispose alors d'une règle spéciale nommée @page, qui désigne la page physique et non un élément du document.

Tableau B–10 Propriétés de pagination en CSS 2.1

<table>
<tr><th>Propriété / pseudoclasse</th><th>Description</th></tr>
<tr><td>:first, :left, :right</td><td>Pseudoclasses utilisables sur @page pour régler par exemple les marges indépendamment pour la première page, les pages gauches (paires pour un document de gauche à droite) et les pages droites (impaires ibidem).</td></tr>
<tr><td>margin
-[top|right|
bottom|left]</td><td>Marges classiques ; je les remets ici car elles ont un sens particulier lorsqu'on les applique à @page : ce sont alors les marges d'impression.</td></tr>
</table>

Tableau B–10 Propriétés de pagination en CSS 2.1 (suite)

Propriété / pseudoclasse	Description
`orphans`	Nombre minimum de lignes d'un bloc qui doivent apparaître en bas de page (lignes orphelines). Si le haut de la page est trop plein, le bloc démarre à la page suivante. La valeur est numérique, et vaut par défaut 2.

`page-break-after`
`page-break-before`
`page-break-inside`
	Régissent les sauts de page. Appliquées à un élément, elles déterminent ce que le navigateur a le droit de faire après, avant et à l'intérieur de l'élément, respectivement. Les deux premières peuvent valoir `auto` (défaut), `always` (force le saut), `avoid` (éviter à tout prix), `left` ou `right` (forcer le saut jusqu'à une page gauche ou droite, par exemple pour un début de chapitre). La dernière ne peut valoir que `auto` ou `avoid`.

`widows`	Nombre minimum de lignes d'un bloc qui doivent apparaître en haut de page (lignes veuves). S'il ne reste pas assez de lignes dans le bloc, il démarre à cette page. La valeur est numérique, et vaut par défaut 2.

Propriétés de couleurs et d'arrière-plan

Tableau B–11 Propriétés de couleurs et d'arrière-plan en CSS 2.1

Propriété	Description
`background` *`-[color|image|repeat|attachment|position]`*	Définit l'arrière-plan d'un élément. On a d'abord sa couleur, puis une image à utiliser, son mode de répétition (« mosaïque » : `repeat` par défaut, mais connaissez-vous `repeat-x`, `repeat-y` et `no-repeat` ?), son mode de défilement (`scroll` par défaut, mais connaissez-vous `fixed` ?) et sa position initiale dans l'élément (par exemple `top right`, ou `15% bottom`, ou `2cm top`).
`color`	Couleur du texte.

Propriétés de gestion de la police de caractères

Tableau B–12 Propriétés de la police de caractères en CSS 2.1

Propriété	Description
`font` *`-[style|variant|weight|size|family]`*	Régit la police de caractères. Voilà un cas où bien apprendre la syntaxe consolidée de la propriété courte (`font`) est payant ! Le style est généralement `normal` ou `italic`, la variante `normal` ou `small-caps`, le poids `normal` ou `bold` (presque aucun système de polices ne gère plus de 2 degrés de graisse...), la taille a une syntaxe plus complexe (voir ci-après) et la famille aussi. La propriété courte peut aussi utiliser juste un nom réservé de police système.

Taille de police

La taille peut être exprimée avec un mot-clé absolu, un mot-clé relatif, une taille classique ou un pourcentage de la taille de référence.

- Absolus : `xx-small`, `x-small`, `small`, `medium` (défaut), `large`, `x-large`, `xx-large`. Le rapport entre les valeurs successives n'est pas fixe, en particulier aux extrêmes.
- Relatifs : `smaller`, `larger`. Décale la taille sur l'échelle des absolus, et si on est déjà sur un extrême, interpole au mieux.

Dans le cas où la taille est précisée au sein de la propriété courte `font`, on dispose d'une syntaxe spéciale qui permet de faire d'une pierre deux coups en précisant à la volée le `line-height` : on utilise `font-size/line-height`, c'est très pratique et cohérent. Voir l'exemple un peu plus loin.

Famille de polices

La famille de polices permet de définir une série de polices à tenter d'utiliser, par ordre décroissant de préférence. Il s'agit de noms de polices que le navigateur va chercher sur le système de l'internaute. Les noms à espaces doivent être entre guillemets. Il est fortement conseillé, pour des raisons d'accessibilité, de terminer la série par un des noms génériques :

- `serif` : police à empattements, par exemple Times ;
- `sans-serif` : police sans empattements, par exemple Arial ;
- `cursive` : police à pleins et déliés, par exemple Monotype Corsiva ou Zapf Chancery ;
- `fantasy` : police « délirante », décalée, amusante ;
- `monospace` : police à chasse fixe, par exemple Courier.

Voici un exemple :

```
font-family: "Bitstream Vera Sans Mono", Monaco, monospace;
```

Tout spécifier d'un coup !

Enfin, voici un premier exemple de propriété courte, qui résume tout ce qu'on a besoin de dire sur la police :

```
font: 115%/1.4em "Bitstream Vera Sans Mono", Monaco, monospace
```

Ici, on ne précise ni le style, ni la variante, ni le poids, mais directement la taille (`115 %`), la hauteur de ligne (`1.4em`) et la famille.

Attention : certaines configurations partielles de style, variante et poids peuvent constituer une ambiguïté : ainsi, si vous n'en précisez que une ou deux et utilisez la valeur `normal` pour la dernière, comment savoir de quelle propriété on parle ? Il faut alors être explicite, quitte à utiliser `inherit` pour maintenir les valeurs des propriétés qu'on ne souhaite pas affecter.

```
font: normal 1.5em/1.8em sans-serif;
```

C'est ambigu : est-ce le type, la variante ou le poids qui est normal ?

```
font: italic normal inherit/120%;
```

Et là, est-ce la variante ou le poids ?

```
font: italic inherit normal inherit/120%;
```

Ici, pas de doute : c'est le poids, et on ne touche pas à la variante.

Dernier point : les noms réservés de polices système, qui configurent toute la police d'un coup. Cela permet de réaliser une interface cohérente avec celle de système d'exploitation (ce qui permet, par exemple, de faire des bibliothèques comme *Prototype Windows*). Les valeurs possibles sont :

- `caption`, utilisée pour les contrôles (composants visuels) à libellés (par exemple les boutons, les listes déroulantes).
- `icon`, utilisée pour labéliser les icônes (par exemple sur le bureau).
- `menu`, utilisée dans les menus (barres ou surgissants).
- `message-box`, utilisée pour les boîtes de dialogue à message.
- `small-caption`, utilisée pour labéliser les petits contrôles (comme les boutons de barres d'outils).
- `status-bar`, utilisée par les barres d'état.

Il suffit donc d'utiliser par exemple :

```
div#status { font: status-bar; }
```

Pour avoir un élément avec la fonte exacte des barres d'état sur le système de l'internaute.

Propriétés de gestion du corps du texte

Tableau B–13 Propriétés du corps du texte en CSS 2.1

Propriété	Description
`letter-spacing`	Espacement entre les lettres. Vaut zéro par défaut. Je conseille vivement de n'utiliser que des tailles en unité `ex`, très adaptée. Rien qu'à `0.1ex`, on voit l'effet.
`text-align`	Alignement du texte dans un bloc. Peut valoir `left`, `center`, `right` ou `justify`.
`text-decoration`	Régit l'apparence de traits au-dessus ou au-dessous du texte. Peut valoir `none` (défaut), `underline` (soulignement), `overline` (trait au-dessus du texte), `line-through` (texte barré) ou… oserai-je le dire ? Bon, `blink`, mais gare au premier qui s'en sert ! C'est moche et tout le contraire d'accessible !
`text-indent`	Indentation de la première ligne d'un bloc de texte.
`text-transform`	Gère la casse. Peut valoir `none` (défaut), `capitalize` (initiales en majuscules, autres lettres toutefois inchangées), `uppercase` (majuscules) ou `lowercase` (minuscules). Voir aussi `font-variant` dans le tableau B-12.
`white-space`	Gestion des espacements dans le corps du texte. Voir ci-après.
`word-spacing`	Espacement entre les mots. Même remarque que pour `letter-spacing`.

L'espacement dans le corps du texte

La propriété `white-space` mérite tout de même une petite explication.

En temps normal, le *rendering* d'un contenu texte retire tous les espacements (espaces, tabulations, retours chariot, etc.) au début et à la fin du texte, ramène toute autre série d'espacements à une seule espace classique (y compris les retours chariot), et va à la ligne quand c'est nécessaire (quand le texte arrive en bout de largeur du bloc conteneur).

On a donc trois comportements distincts : la réduction des espacements, le respect des retours chariot d'origine et le passage à la ligne pour honorer la largeur du conteneur (*wrapping*). Voici les définitions succinctes des valeurs possibles pour `white-space` !

Tableau B–14 Valeurs de white-space

Valeur	Réduction	Retours chariots	Wrapping
`normal`	Oui	Non	Oui
`pre`	Non	Oui	Non
`nowrap`	Oui	Non	Non
`pre-wrap`	Non	Oui	Oui
`pre-line`	Oui	Oui	Oui

Propriétés des tableaux

Tableau B–15 Propriétés des tableaux en CSS 2.1

Propriété	Description
`Border-collapse`	Gère la fusion des bordures entre cellules adjacentes, et entre les cellules et la bordure de la table. Désactivée par défaut (`separate`), peut être activée avec la valeur `collapse`. Je trouve ça beaucoup plus joli, personnellement…
`border-spacing`	Espacement entre bordures de cellules (si les bordures ne sont pas fusionnées). Peut contenir une ou deux tailles. Dans le second cas, distingue entre distances horizontale et verticale.
`caption-side`	Position du titre : au-dessus (`top`, défaut) ou au-dessous (`bottom`).
`empty-cells`	Affiche ou masque les cellules vides. Par défaut, affiche (`show`). On les masque avec `hide`.
`table-layout`	Mode de calcul des largeurs du tableau et des cellules. Le mode par défaut, `auto`, est celui auquel on s'attend : il adapte les largeurs en fonction des contenus de cellules. L'autre mode, `fixed`, utilise uniquement les spécifications de largeur pour le tableau, les colonnes, les bordures et l'espacement entre cellules. Il est plus rapide mais rend généralement moins bien.

Propriétés de l'interface utilisateur

Tableau B–16 Propriétés de l'interface utilisateur en CSS 2.1

Propriété	Description
`cursor`	Détermine le curseur souris à utiliser quand celui-ci survole l'élément. Extrêmement utile en terme d'ergonomie. Les valeurs sont détaillées à la section 18.1 de la recommandation, mais je cite les principales : `auto` (défaut), `default` (curseur classique du système d'exploitation), `pointer` (comme pour un lien), `move` (idéal pour glisser-déplacer), `help` (idéal pour `abbr` et `acronym`).
`outline` *`-[color|style|width]`*	Affiche une délimitation autour d'un élément. Diffère d'une bordure en ce qu'elle n'occupe pas de place dans le modèle des boîtes : elle est dessinée au-dessus du bord extérieur de la bordure. Elle ne comprend donc pas les marges. Peut être utile pour signaler qu'un élément est prêt à recevoir un dépôt lors d'un glisser-déplacer, mais encore mal pris en charge…

Pour aller plus loin…

Livres

CSS 2 – Pratique du design web
Raphaël Goetter
Eyrolles, juin 2005, 324 pages (de bonheur)
ISBN 2-212-11570-9

Le zen des CSS
Dave Shea
Eyrolles, novembre 2005, 296 pages
ISBN 2-212-11699-3

Cascading Style Sheets: The Definitive Guide
Eric Meyer
O'Reilly, novembre 2005, 508 pages
ISBN 0-596-00525-3

Mémento CSS
Raphaël Goetter
Eyrolles, novembre 2005, 14 pages (de mémento)
ISBN 2-212-11726-4

Sites

- La recommandation CSS 2.1, évidemment. Attention, seule la version anglaise a valeur de référence :
  - http://www.w3.org/TR/CSS21/
  - Version française de la 2.0 : http://www.yoyodesign.org/doc/w3c/css2/cover.html
- L'excellent site géré par Raphaël Goetter (jetez-vous sur son livre !), Alsa Créations : http://www.alsacreations.com/
- Le CSS Zen Garden, pour se convaincre qu'avec le même XHTML, on peut changer complètement de tête : http://csszengarden.com/
- A List Apart brille aussi en CSS : http://alistapart.com/
- Des styles décalés, les frontières de l'impossible : CSS Play. http://moronicbajebus.com/playground/cssplay/
- Roger Johansson a plein de choses à vous raconter sur CSS, si vous allez au 456 Berea St. : http://www.456bereastreet.com/
- CSS Beauty : http://www.cssbeauty.com/



Le « plus » de l'expert : savoir lire une spécification

Il existe trois catégories de développeurs web. D'abord ceux qui semblent toujours tout savoir, quitte à ne vous répondre que quelques instants plus tard, et expriment leur réponse avec un air d'autorité confiante dans l'exactitude de leurs propos, laquelle se vérifie en effet à chaque fois. Ensuite ceux qui n'ont pas toutes les réponses, et semblent ne pas trop savoir où les chercher. Enfin, ceux qui manifestement n'ont qu'une compétence empirique : leurs pages « tombent en marche ».

Vous souhaitez évidemment ne pas faire partie de la dernière catégorie, ni même avoir à travailler avec de telles personnes. Vous connaissez probablement un certain nombre de développeurs entrant dans la deuxième catégorie ; c'est peut-être d'ailleurs votre cas. Quant à ceux de la première catégorie, ces puits apparemment sans fond de connaissances, ils suscitent l'admiration de tous. Cette annexe vous propose d'en faire partie.

Intérêts d'aller chercher l'information à la source

Il y a deux intérêts fondamentaux à être capable d'aller chercher l'information à la source. Le premier est parfaitement objectif et professionnel. Le second est plus subjectif et, comment dire… humain.

Certitude et précision

Les bonnes spécifications ont plusieurs qualités. Intrinsèquement d'abord, elles constituent le document de référence pour une technologie : elles font donc autorité sur la question. Utiliser correctement la technologie revient à l'utiliser conformément à la spécification. Si cela ne fonctionne pas alors que c'est exactement comme la spécification le demande, on sait que c'est notre environnement de travail qui est fautif et non notre code.

Bien sûr, cela peut simplement vouloir dire qu'on utilise du CSS 2.1 sur MSIE 6, auquel cas on ne peut pas laisser les choses telles quelles, il faudra trouver un compromis.

Une bonne spécification est par ailleurs précise : elle doit indiquer tous les cas particuliers, toutes les nuances, tous les problèmes potentiels. Elle ne doit pas laisser de zones d'ombres. Généralement, cela signifie que la spécification est aride, ou en tout cas particulièrement verbeuse. Les excellentes spécifications arrivent à conjuguer précision totale et lisibilité.

Quiconque maîtrise un sujet sur le bout des doigts le sait bien : il est très agréable de discuter de quelque chose qu'on connaît parfaitement ; en particulier s'il s'agit d'aider quelqu'un à comprendre, à utiliser, à mettre au point. L'expertise est agréable. Être véritablement spécialiste d'un domaine précis et mettre cette expertise en œuvre est très agréable.

Savoir utiliser les spécifications d'une technologie fournit ce que les moyens de deuxième main (livres, didacticiels, articles, ateliers, etc.) ne peuvent que difficilement donner : l'accès à une maîtrise totale, ou à tout le moins l'accès à l'information totale.

« On m'a dit que là-dessus, c'est toi qui sais tout » : l'expertise

Il existe un deuxième avantage, plus humain celui-là. À force de faire preuve d'expertise sur un sujet donné, vous allez être connu pour cela. Un cercle toujours plus large de collègues et connaissances va faire l'association d'idée entre ce sujet et vous. Et de plus en plus, lorsqu'on aura besoin d'une information précise, pointue, fiable, on viendra vous voir.

Être un expert en XHTML, en balisage sémantique, en accessibilité, en CSS 2.1, en DOM niveau 2, en JavaScript et en AJAX ne vous apportera pas fortune et gloire (quoique...), mais dans votre travail, cela risque fort de vous apporter autre chose :

Vous allez devenir indispensable.

Rien que pour l'ego, c'est agréable. Mais cela peut aussi affecter vos prétentions salariales, embellir votre CV et vous ouvrir de nouvelles opportunités.

Les principaux formats de spécifications web

Dans le cadre des technologies web, les spécifications utilisent essentiellement quatre formats.

Les recommandations du W3C

Les technologies gérées par le W3C sont publiées sous formes de recommandations. On trouve deux abréviations courantes : TR (*Technical Report*) et REC (*Recommendation)*. Il s'agit du statut finalisé d'une spécification, qui passe auparavant par plusieurs stades WD (*Working Draft*, ou ébauches).

La plupart des « langages descriptifs » du Web sont des technologies W3C. Citons principalement (X)HTML, XML, CSS, DOM, MathML, RDF, SMIL, SOAP, SVG, XPath et XSL/XSLT.

Les grammaires formelles de langages à balises : DTD et schémas XML

Les langages à balises disposent d'une grammaire formelle, très pratique pour retrouver rapidement le détail des attributs et éléments autorisés dans un contexte précis.

Suivant le cas (principalement l'origine et l'ancienneté du langage visé), la grammaire utilise soit une DTD (*Document Type Definition*), qui est un document SGML de syntaxe assez facile, soit un schéma XML, un document... XML, potentiellement plus puissant mais souvent très, très verbeux.

Par exemple, HTML et XHTML 1.0 utilisent des DTD, tandis que XHTML 1.1+, WSDL et XLink utilisent des schémas XML.

Il est à noter qu'un juste milieu existe au travers de la syntaxe Relax NG. Bien que celle-ci gagne chaque jour en popularité, elle n'est pas encore adoptée par les principaux organismes de standardisation, notamment le W3C.

Les RFC de l'IETF : protocoles et formats d'Internet

Enfin, la plupart des protocoles et formats de données du Web sont gérés et normalisés par l'IETF (*Internet Engineering Task Force*), un très large groupement de professionnels qui est, véritablement, à l'origine d'Internet (premiers standards en 1969 !).

Les standards de l'IETF sont collectivement appelés les RFC (*Request For Comments*), et utilisent un format texte en 72 colonnes, très simple à lire. Il en existe plus de 4 600, dont plus de 300 rien qu'entre janvier et août 2006.

On y trouve notamment les spécifications pour HTTP, SMTP, POP, IMAP, FTP, Telnet, ICMP (la commande `ping`), SSL...

S'y retrouver dans une recommandation W3C

Commençons par explorer la structure d'une recommandation W3C. Le site officiel du W3C est http://w3.org. Vous y trouverez toutes les spécifications dans leur version anglaise, seule à être garantie : des traductions existent souvent, mais leur qualité n'est pas validée en détail par le W3C, même s'ils lient sur ces traductions depuis la version originale.

URL et raccourcis

Les recommandations ont souvent une URL assez longue, car elle contient quelques répertoires et surtout un horodatage de version. Voici quelques exemples, qui donnent une idée des dégâts :

- http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/
- http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/
- http://www.w3.org/TR/2002/CR-css-mobile-20020725
- http://www.w3.org/TR/2001/REC-xhtml11-20010531/
- http://www.w3.org/TR/2003/REC-SVG11-20030114/
- http://www.w3.org/TR/1999/REC-xpath-19991116

Vous y observez le préfixe `/TR/` en début de chemin, comme pour toutes les recommandations. On a ensuite l'année de parution, puis le chemin des documents de la spécification, qui démarre généralement par `REC` (on a ici aussi `CR`, pour *Candidate Recommendation*, dernier stade avant finalisation. On peut aussi trouver `WD` : *Working Draft*, ou `NOTE`, pour les documents à valeur informative).

Les chemins des spécifications précisent toujours la date exacte de parution du document à la fin, au format `aaaammjj`. On peut donc déduire, par exemple, que la der-

nière version du DOM niveau 2 HTML date du 19 janvier 2003, alors que celle du DOM niveau 2 noyau date du 13 novembre 2000.

Pour de nombreuses spécifications, il existe toutefois une URL « raccourcie », qui amène automatiquement à la dernière version. Voici les équivalents des URL mentionnées plus haut, avec quelques autres :

- http://www.w3.org/TR/html401/
- http://www.w3.org/TR/xhtml1/
- http://www.w3.org/TR/CSS21/
- http://www.w3.org/TR/DOM-Level-2-Core/
- http://www.w3.org/TR/DOM-Level-2-HTML/
- http://www.w3.org/TR/css-mobile
- http://www.w3.org/TR/xhtml11/
- http://www.w3.org/TR/SVG11/
- http://www.w3.org/TR/xpath

En fait, cela revient le plus souvent à supprimer l'année, le préfixe REC et l'horodatage en fin de nom. Notez qu'il y a parfois des *slashes* (/) terminaux, et parfois non. Dans certains cas (HTML 4.01, XHTML 1.0, CSS 2.1...) cela n'a aucune importance, dans d'autres vous obtiendrez une page intermédiaire.

Structure générale d'une recommandation

Une recommandation W3C a toujours la même structure générale.

D'abord l'en-tête :

1. titre avec version ;
2. statut (recommandation, ébauche...) et date de publication ;
3. liste de liens vers les formats disponibles (texte, HTML, PDF...) ;
4. liens vers la dernière version et la version précédente ;
5. liste des éditeurs, c'est-à-dire des responsables de la spécification.

La figure C–1 illustre l'en-tête de la recommandation DOM niveau 2 HTML.

Comme vous le voyez, la structure n'est pas toujours exactement identique à celle décrite plus haut, mais on retrouve très vite ses repères : ici, les formats sont simplement listés après la liste des éditeurs. Le lien sur les traductions, qui figure souvent après la partie introductive, est ici en fin d'en-tête. Mais à part ça, on reste dans le moule. Par contraste, observez l'en-tête de la recommandation pour HTML 4.01, qui correspond à l'ancienne façon de faire, et reprend exactement notre liste (figure C–2).

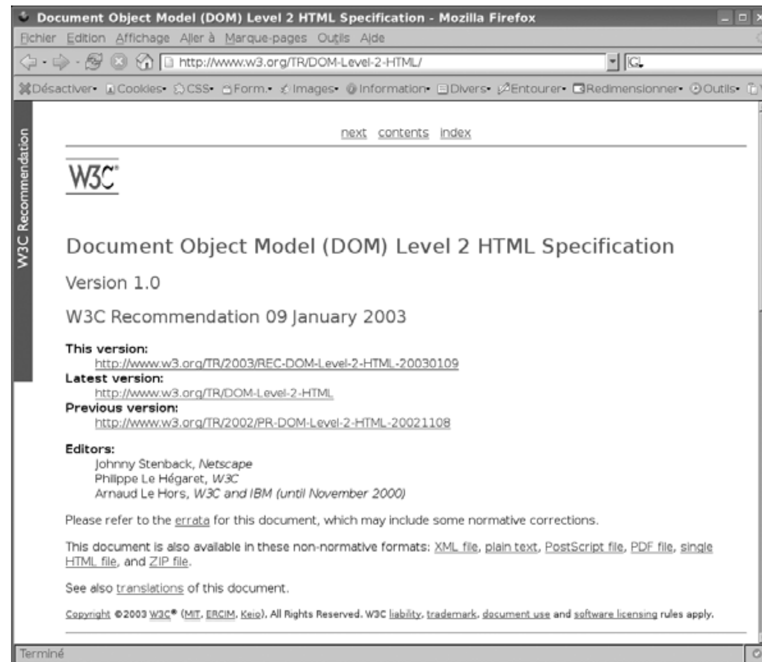


Figure C–1
L'en-tête de la recommandation DOM niveau 2 HTML

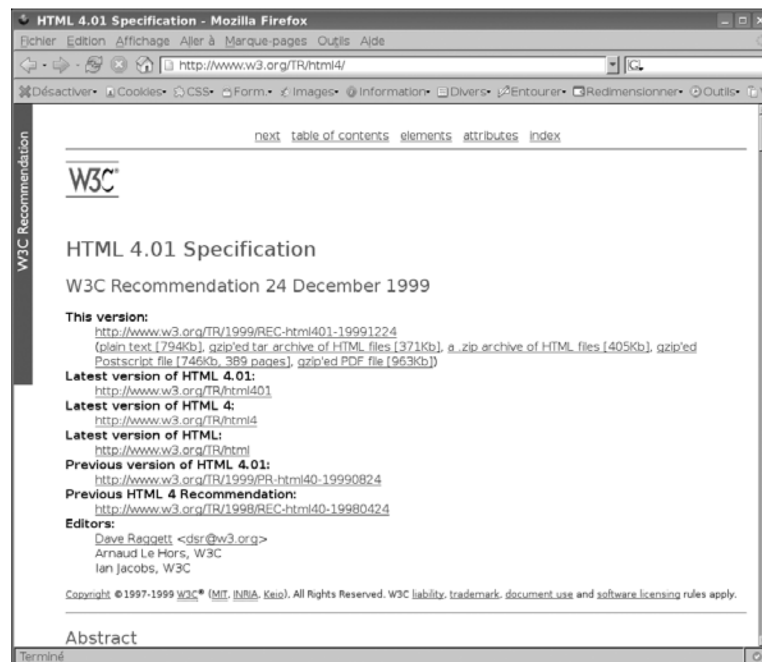


Figure C–2
L'en-tête de la recommandation HTML 4.01

Il a même une particularité, qui est de proposer les versions à jour et précédentes pour les variantes 4 et 4.01. Notez aussi que la liste des formats était bien moins lisible que la forme adoptée plus récemment.

On trouve ensuite la partie introductive, qui fournit :

1 L'*abstract*, qui décrit rapidement le rôle de la technologie spécifiée.
2 Le statut du document, qui contient toujours un texte plus ou moins pro forma sur le statut (recommandation, ébauche, etc.), l'état non normatif s'il s'agit d'une traduction, et fournit la liste des traductions connues (ou en tout cas un lien dessus, chercher le lien *translations* dans le corps du texte faute d'une section *Available languages*) ainsi qu'un lien vers les corrections ultérieures à la publication (errata).
3 La table des matières.

Suivant la spécification, on a alors plusieurs possibilités :

- Toute la spécification est sur la même page, le même document (cas de XHTML 1.0 ou XPath, par exemple).
- Seule la table des matières y figure et chaque section a une page dédiée (c'est le cas par exemple de HTML 4.01, XHTML 1.1 et CSS 2.1).
- La table des matières présentée ne garde que les parties incontournables (table des matières justement, copyright, annexes classiques, glossaire, références, index) et une seule section (deux tout au plus) qui constitue le cœur du sujet, mais figure dans un sous-document (cas quasi systématique dans les spécifications du DOM, voir figure C-3).

On reconnaît vite la nature du document, rien qu'à la taille du curseur dans la barre de défilement verticale : s'il est très petit, il est probable qu'on a toute la spécification sur une seule page !

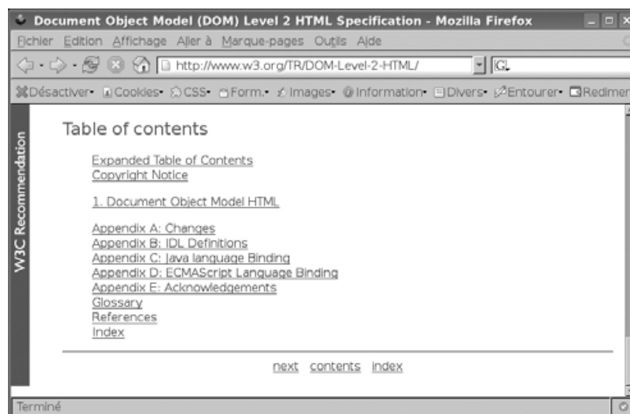


Figure C–3
Spécifications DOM : une table des matières courte et un sous-document

Le format premier d'une recommandation est le HTML. Les documents utilisent donc abondamment les hyperliens, ce qui rend leur consultation plus pratique. Toutefois, la simple taille des recommandations fait qu'on s'y perd facilement. Pour s'y retrouver, on a deux moyens. Si la spécification ne comporte que quelques pages (voire une seule), la recherche interne au navigateur doit permettre de s'y retrouver rapidement. On l'active généralement avec `Ctrl+F`.

Pour des spécifications plus distribuées, comportant de nombreuses pages, il est bon de regarder si la recommandation fournit effectivement un index, qui figure généralement tout en bas de la table des matières. Chaque terme important (notamment tous les noms d'éléments, de propriétés, de méthodes, d'interfaces, etc.) fait l'objet d'un lien ; en cas de liens multiples, le premier porte le nom de l'élément et les suivants des numéros.

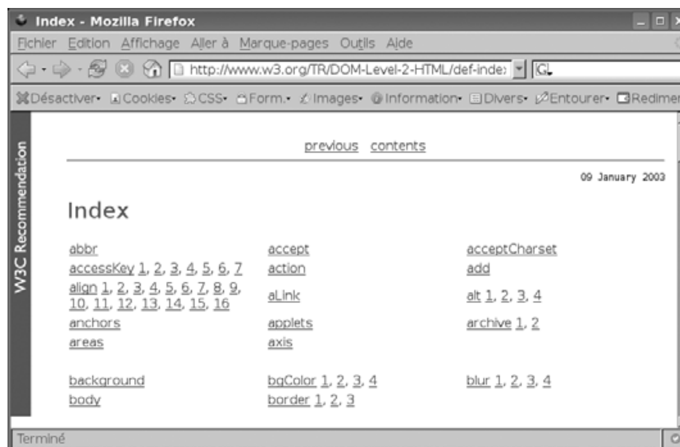


Figure C–4
Index d'une recommandation W3C

Astuce utile lorsqu'on cherche à déterminer si un aspect précis appartient à une version donnée ou à la précédente (ou simplement pour examiner rapidement en quoi la nouvelle version diffère) : chaque recommandation dispose en annexes (et même dans les toutes premières annexes) d'une liste des changements. Suivant la taille de la recommandation, cette annexe est elle-même plus ou moins structurée. Ainsi, les changements pour CSS 2.1 (en réalité pour CSS 2.0 et 2.1) sont impressionnants (figure C–5).

En revanche, dans le DOM niveau 2 HTML, on est plus sobre : une simple entrée comme annexe A, nommée *Changes*. Il faut dire que le document détaillant les changements depuis DOM niveau 1 pour les éléments relatifs à HTML occupe à peine deux écrans de haut.

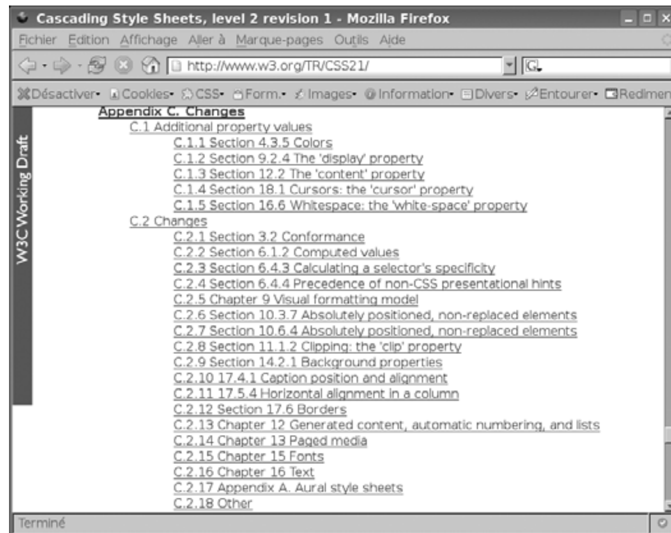


Figure C–5
Que de changements entre CSS 1 et CSS 2.1 !

Recours à des syntaxes formelles

Suivant ses besoins, une recommandation W3C va s'appuyer sur des syntaxes formelles adaptées pour décrire des aspects techniques. Les principales syntaxes employées sont :

- DTD pour les éléments de balisage jusqu'à XHTML 1.0 Strict.
- Schéma XML à partir de XHTML 1.1.
- IDL (*Interface Description Language*) pour les interfaces (essentiellement dans le cadre du DOM).
- EBNF (*Extended Backus-Naur Form*), Lex ou Yacc pour le reste.

En raison de leur fréquence, nous étudierons plus en détail DTD et schéma XML dans les sections qui suivront.

IDL est très facile à lire, car elle ressemble à une déclaration de classe abstraite ou d'interface dans les principaux langages objets. C'est parfois aussi simple que dans le code suivant.

Listing C-1 Déclaration IDL de l'interface HTMLElement (DOM niveau 2 HTML)

```
interface HTMLElement : Element {
           attribute DOMString       id;
           attribute DOMString       title;
           attribute DOMString       lang;
           attribute DOMString       dir;
           attribute DOMString       className;
};
```

C'est parfois un peu plus compliqué, mais ça reste facilement lisible.

Listing C-2 Déclaration IDL de l'interface HTMLSelectElement

```
interface HTMLSelectElement : HTMLElement {
  readonly attribute DOMString       type;
           attribute long            selectedIndex;
           attribute DOMString       value;
  // Modified in DOM Level 2:
           attribute unsigned long   length;
           // raises(DOMException) on setting

  readonly attribute HTMLFormElement form;
  // Modified in DOM Level 2:
  readonly attribute HTMLOptionsCollection options;
           attribute boolean         disabled;
           attribute boolean         multiple;
           attribute DOMString       name;
           attribute long            size;
           attribute long            tabIndex;
  void               add(in HTMLElement element,
                         in HTMLElement before)
                             raises(DOMException);
  void               remove(in long index);
  void               blur();
  void               focus();
};
```

Quant à EBNF, il s'agit d'une des plus anciennes syntaxes textuelles de description de grammaire. Vous trouverez quelques explications sur cette syntaxe, plutôt simple, aux deux URL suivantes :

- http://developpeur.journaldunet.com/tutoriel/theo/050831-notation-bnf-ebnf.shtml
- http://fr.wikipedia.org/wiki/EBNF

Certaines spécifications utilisent des descriptions plus basées sur les syntaxes Lex (ou Flex) et Yacc (ou Bison), bien connues des développeurs C. Ce n'est pas très éloigné d'EBNF. Voici un exemple tiré de la recommandation CSS 2.1, qui repose lui-même sur quelques définitions mentionnées plus haut dans le document.

Listing C-3 Définition Lex de syntaxe générale pour une feuille de styles CSS

```
stylesheet  : [ CDO | CDC | S | statement ]*;
statement   : ruleset | at-rule;
at-rule     : ATKEYWORD S* any* [ block | ';' S* ];
block       : '{' S* [ any | block | ATKEYWORD S* | ';' S* ]* '}' S*;
ruleset    : selector? '{' S* declaration? [ ';' S* declaration? ]* '}' S*;
```

```
selector    : any+;
declaration : DELIM? property S* ':' S* value;
property    : IDENT;
value       : [ any | block | ATKEYWORD S* ]+;
any         : [ IDENT | NUMBER | PERCENTAGE | DIMENSION | STRING
              | DELIM | URI | HASH | UNICODE-RANGE | INCLUDES
              | DASHMATCH | FUNCTION S* any* ')'
              | '(' S* any* ')' | '[' S* any* ']' ] S*;
```

Il faut bien comprendre que, l'immense majorité du temps, des descriptions EBNF, Lex ou Yacc visent plus les développeurs de logiciels implémentant la technologie que ceux qui utilisent cette même technologie. Le texte environnant fournit généralement tous les détails nécessaires pour votre utilisation.

Les descriptions de propriétés CSS

La section *Déchiffrer une DTD*, plus loin dans cette annexe, montrera comment lire et exploiter les fragments de DTD employés pour décrire les langages à balises, par exemple HTML. Avant d'en finir avec les recommandations W3C, je voudrais tout de même donner quelques informations supplémentaires sur deux formats très consultés dans la pratique : celui décrivant les propriétés CSS, et celui décrivant les propriétés et méthodes du DOM.

Commençons donc par les propriétés CSS. Dans un souci de référence, nous utiliserons la version originale, en anglais. Prenons par exemple la description de la propriété `white-space`, dans la section *Text*. La spécification utilise la représentation suivante pour décrire la propriété :

'white-space'

Value:	normal | pre | nowrap | pre-wrap | pre-line | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed values:	as specified

1 *Value*, détaille la liste des valeurs possibles, séparées par des *pipes* (|), symbole fréquent pour indiquer une alternative. Les valeurs exprimées sont normatives et la casse est parfois significative. On a ici 6 valeurs possibles, dont l'incontournable `inherit` pour une propriété héritable.

2 *Initial* décrit la valeur par défaut, *none* s'il n'y en a aucune.
3 *Applies to* décrit les catégories d'éléments qui disposent de cette propriété. Ici elle s'applique à tous, mais des valeurs courantes sont *block-level elements*, *inline elements*, etc. Il s'agit des catégories déterminées par le modèle visuel CSS, évoqué à l'annexe B.
4 *Inherited* indique si la propriété est héritable ou pas ; une propriété héritable prend sa valeur par défaut dans la valeur pour son élément conteneur : c'est dans le principe de la cascade. Toutes les propriétés ne sont pas héritables (par exemple, `text-decoration` ne l'est pas).
5 *Percentages* est utile quand les valeurs possibles incluent une notation en pourcentage. C'est principalement le cas des tailles (de boîte ou de police de caractères). Cette valeur indique alors à quoi se réfèrent les pourcentages (par exemple, la largeur du bloc conteneur).
6 *Media* indique le ou les média CSS pour lesquels la propriété a du sens. Ici on est sur `visual`, ce qui couvre tous les média visuels : `screen`, `print`, `projection`, etc., tout en excluant les média comme `aural` (lecteurs d'écran) par exemple.
7 *Computed values*, enfin, précise les ajustements à apporter à la valeur en fonction du contexte. C'est parfois simple et donc indiqué à la volée. Quand c'est plus complexe, on a généralement *as specified*, et les détails dans le texte qui suit. Par exemple, la propriété `text-align` varie en valeur suivant ce que vaut la propriété `white-space`.

Les descriptions de propriétés et méthodes DOM

Autre format fréquemment consulté, les propriétés et méthodes DOM. On a déjà illustré le code IDL utilisé pour représenter une interface DOM complète, mais ces fragments de code sont suivis d'explications plus détaillées, bien entendu. Une interface DOM est toujours spécifiée en plusieurs temps :

1 le code IDL de l'interface ;
2 les descriptions des attributs éventuels ;
3 les descriptions des méthodes éventuelles.

Le code IDL fournit quelques informations précieuses. D'abord, au niveau de la déclaration de l'interface elle-même, si celle-ci étend une autre interface, on le voit immédiatement. Quelques exemples :

```
interface HTMLDocument : Document {
...
interface HTMLElement : Element {
...
interface HTMLImageElement : HTMLElement {
...
```

On voit immédiatement qu'un élément `img`, qui expose naturellement l'interface `HTMLImageElement`, expose donc aussi l'interface `HTMLElement`, et donc `Element`, et donc `Node`. Il fournit donc de nombreuses propriétés et méthodes !

Par ailleurs, le code IDL fournit une vue globale sur les types des propriétés et ceux des méthodes (types de retour, types des arguments). Quelques exemples là aussi, au travers du code IDL de `HTMLOptionsCollection`, reformaté :

```
// Introduced in DOM Level 2:
interface HTMLOptionsCollection {
  attribute unsigned long   length;
    // raises(DOMException) on setting
  Node item(in unsigned long index);
  Node namedItem(in DOMString name);
};
```

Que trouve-t-on ici ? D'abord, les attributs sont reconnaissables à ce qu'ils sont préfixés par `attribute`, et n'ont pas de parenthèses après leur nom. Le mot réservé `attribute` est parfois précédé de `readonly`., ce qui est très important : cela signifie qu'il est en lecture seule. Toute tentative d'écriture générera sans doute une exception.

Entre `attribute` et le nom de l'attribut, on a le type. IDL utilise des types primitifs issus du C, et des types objets qui sont soit d'autres interfaces du DOM, soit des types classiques définis par le DOM noyau.

Les méthodes n'ont pas `attribute` au début, mais un type de retour, un nom, et des parenthèses entre lesquelles on peut lister des paramètres, avec leur sens, leur type et leur nom. Toutes ces informations sont précieuses, mais le sens est généralement `in` et peut être ignoré. Il signifie simplement que la méthode utilise l'argument sans le modifier : c'est un paramètre local, passé par valeur, si vous préférez.

Voici une liste des principaux types utilisés dans l'IDL des spécifications DOM.

Tableau C–1 Principaux types utilisés dans l'IDL du DOM

Type	Description
`[unsigned] short`	Nombre entier 16 bits. Soit signé (-32 768 à 32 767), soit non signé (0 à 65 535).
`[unsigned] long`	Nombre entier 32 bits. Même remarque pour le signe. C'est le type numérique le plus courant.
`boolean`	Valeur booléenne : `false` ou `true`.
`void`	Aucune valeur. Type de retour des méthodes ne renvoyant rien.
`DOMString`	Chaîne de caractères Unicode (16-bit).
`DOMTimeStamp`	Nombre entier positif 64 bits représentant un nombre de millisecondes depuis le début de l'ère (1er janvier 1970 00:00:00 GMT). De nombreux langages représentent ainsi les moments.

Tableau C–1 Principaux types utilisés dans l'IDL du DOM (suite)

Type	Description
`DOMException`	Le type standard des exceptions levées par les méthodes. Contient simplement une propriété numérique entière nommée `code`, qui vaut l'une des constantes *xxx*_ERR décrites dans les recommandations du DOM.
Interface DOM (ex. `Node`)	Eh bien, l'interface en question... Reportez-vous à sa définition !

Les valeurs numériques signées sont assez rares : on utilise principalement `unsigned long`.

Après le code IDL, on trouve une section *Attributes* s'il y a des attributs, et une section *Methods*... s'il y a des méthodes.

Une section d'attribut reprend son nom, son type et son éventuelle contrainte de lecture seule. Un texte décrit l'attribut plus en détail, et si celui-ci a des contraintes pour son écriture, un paragraphe particulier précise l'information *Exception on setting* qui figurait déjà, normalement, en commentaire dans le code IDL.

Une section de méthode reprend son nom, puis décrit la méthode plus en détail avec un texte explicatif. Ce texte est obligatoirement suivi de trois sections : les paramètres, la valeur de retour, et les exceptions potentielles. Les méthodes simplissimes se retrouvent donc avec trois embryons de sections. Ce qui donne par exemple :

close

Closes a document stream opened by `open()` and forces rendering.

No Parameters

No Return Value

No Exceptions

Ces sections fournissent souvent le petit détail qui explique pourquoi votre appel produit une erreur, ou ne donne rien, ou encore engendre un avertissement. Bien les lire est précieux.

Quand vous utilisez pour la première fois une interface DOM, lisez sa documentation complète. Ne vous contentez pas de quelques bouts qui semblent répondre à vos questions immédiates. Prenez quelques minutes de plus pour lire l'ensemble, cela vous économisera souvent bien des heures de débogage par la suite.

Déchiffrer une DTD

La DTD (*Document Type Definition*) constitue le premier format historique de grammaire pour les langages à balises. Une DTD est rédigée en SGML (*Standard Generalized Markup Language*), langage dont le HTML est en quelque sorte un sous-ensemble.

Si vous rédigez correctement vos pages web, vous faites référence à une DTD tout au début de votre document, à l'aide d'une instruction `DOCTYPE`. Par exemple, la première ligne de votre page web dit normalement :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Comment ça, « non » ?! Mais c'est mal ! Vous ne faites pas du XHTML 1.0 Strict ? Vous n'avez pas honte ? Allez donc relire l'annexe A et faites pénitence.

Déclarer explicitement sa DTD de référence dans une page web a un double avantage. D'une part, cela permet d'utiliser un validateur HTML correctement : celui-ci pourra détecter votre grammaire au lieu d'essayer de la deviner, et vous assurera ainsi une vérification de grammaire appropriée. D'autre part, déclarer une DTD stricte (qu'elle soit HTML 4.01 ou XHTML 1.0) permet à votre navigateur de passer en mode « respect des standards », notamment en ce qui concerne les CSS. Pour plus de détails, cherchez donc la phrase *doctype switching* sur Google.

Mais revenons à nos DTD dans le cadre de spécifications. Une DTD constitue une spécification formelle pour un langage à balises (par exemple, HTML 4.01). Elle n'explique pas le sens des balises ou des attributs, mais décrit quelles balises et quels attributs sont disponibles, où et quand.

Les spécifications HTML et apparentées ont pris l'habitude, au début de chaque section décrivant un élément, de présenter le fragment correspondant de la DTD. Observez par exemple la spécification de HTML 4.01 (sur laquelle repose XHTML 1.0, ne l'oubliez pas) pour l'élément `form`.

Listing C-4 Le fragment de la DTD HTML 4.01 pour form

```
<!ELEMENT FORM - - (%block;|SCRIPT)+ -(FORM) -- interactive form -->
<!ATTLIST FORM
  %attrs;                               -- %coreattrs, %i18n, %events --
  action      %URI;          #REQUIRED -- server-side form handler --
  method      (GET|POST)     GET       -- HTTP method used to submit the form --
  enctype     %ContentType;  "application/x-www-form-urlencoded"
  accept      %ContentTypes; #IMPLIED -- list of MIME types for file upload --
  name        CDATA          #IMPLIED -- name of form for scripting --
  onsubmit    %Script;       #IMPLIED -- the form was submitted --
  onreset     %Script;       #IMPLIED -- the form was reset --
  accept-charset %Charsets;  #IMPLIED -- list of supported charsets --
  >
```

On a ici la description de l'élément form lui-même, avec son nom et la description de son contenu. Vous remarquez sans doute qu'ici, les noms d'éléments sont en majuscules, « à l'ancienne ». C'était la norme du temps de HTML (au siècle dernier, donc), mais depuis que XHTML est arrivé, on est sensible à la casse et les nouvelles normes officialisent la casse minuscule.

Les recommandations suivent d'ailleurs les pratiques de leur temps : même s'il est aujourd'hui communément admis que XHTML est incontournable, et qu'il faut donc respecter les règles de fermeture de balises, de valeurs d'attributs entre guillemets, et de balises en minuscules, la recommandation HTML foisonne toujours d'exemples « d'époque », pour ainsi dire, où beaucoup d'éléments ne sont pas fermés, sont en majuscules et n'encadrent pas leurs valeurs d'attributs !

Détaillons rapidement la syntaxe de notre fragment. On a d'abord :

```
<!ELEMENT FORM - - (%block;|SCRIPT)+ -(FORM) -- interactive form -->
```

Le début, <!ELEMENT, indique qu'on définit un élément. Le nom suit : FORM.

Après les deux tirets séparés, on trouve une description du modèle de contenu : (%block;|SCRIPT)+ -(FORM). On peut traduire ce modèle comme ceci : « soit le modèle *block*, soit l'élément SCRIPT, le tout autant de fois qu'on veut (mais au moins une fois), en revanche, pas d'élément FORM imbriqué ».

En effet, %block est ce qu'on appelle une référence d'entité, sur laquelle il est d'ailleurs possible de cliquer dans la recommandation. La définition de l'entité %block la décrit comme pouvant être réalisée par un certain nombre de balises connues, dont par exemple P, H1, UL, PRE, DIV, mais aussi FORM, ce qui amène justement l'exclusion explicite dans notre définition d'élément FORM.

La dernière partie, -- interactive form --, constitue un commentaire de fin de définition, qui nous signale que l'élément FORM permet de réaliser un formulaire utilisateur.

Passons maintenant à la liste des attributs pour l'élément FORM. Classiquement, elle figure dans la DTD juste après la définition de l'élément lui-même. J'ai retiré les alignements entre les champs pour améliorer la lisibilité individuelle des extraits. La liste des attributs débute par :

```
<!ATTLIST FORM
  %attrs; -- %coreattrs, %i18n, %events --
```

On déclare ici une définition d'attributs pour l'élément FORM. Les premiers attributs sont référencés par l'entité %attrs, dont le commentaire nous apprend qu'elle représente une combinaison des entités %coreattrs, %i18n et %events.

Ces entités regroupent les attributs dits noyau (`id`, `class`, `style` et `title`), ceux relatifs à la gestion des langues (`lang` et `dir`) et ceux relatifs aux événements JavaScript (par exemple `onclick`). Vous n'utiliserez bien entendu jamais ces derniers, puisque vous faites de l'*unobstrusive JavaScript*, n'est-ce pas ?

Que dit la suite ?

```
action %URI; #REQUIRED -- server-side form handler --
```

On a là un attribut `action`, dont le modèle de contenu est référencé par l'entité `%URI` (laquelle, idéalement, devrait décrire la syntaxe d'un URI, mais la syntaxe DTD étant pauvre, elle se contente, faute de mieux, de dire simplement « texte quelconque »...). On précise aussi que cet attribut est obligatoire (et je me suis rendu coupable d'enfreindre cette règle dans certains exemples JavaScript de ce livre, je l'avoue...).

Deux autres exemples intéressants de définition d'attributs :

```
method (GET|POST) GET -- HTTP method used to… --
name    CDATA       #IMPLIED -- name of form for scripting --
```

On a ici une autre forme de modèle de contenu, qui dit en substance : « l'attribut `method` peut valoir `GET` ou `POST`, mais pas autre chose » (ce qui embête tant aujourd'hui les partisans d'une approche REST pour les API web). On voit aussi une valeur par défaut au lieu de `#REQUIRED` : si la méthode n'est pas précisée, elle vaudra `GET`.

La définition de l'attribut `name` indique qu'il s'agit d'un texte quelconque (type spécial `CDATA`, pour *character data*), et qu'il est optionnel sans valeur par défaut (`#IMPLIED`).

Voilà pour une introduction suffisante. Astuce intéressante : la spécification HTML 4.01 a jugé bon de fournir une présentation plutôt détaillée des syntaxes DTD utilisées, et cela directement dans la recommandation ! Voici l'adresse de la version française afin de vous faciliter le plus possible l'acquisition : http://www.la-grange.net/w3c/html4.01/intro/sgmltut.html#h-3.3. Si cela ne suffit pas, vous avez une introduction générique assez bien faite qui pourra peut-être mieux vous satisfaire sur http://www.w3schools.com/dtd/default.asp.

J'ai dit tout à l'heure que la DTD précisait quelles balises et quels attributs étaient disponibles, où et quand. Dit comme cela, on a l'impression que toute l'information peut être transmise, et que le corps du texte de la recommandation n'est là que pour préciser le sens des éléments et des balises et ajouter quelques informations de contexte.

En réalité, une DTD n'est pas très précise. La syntaxe disponible ne permet pas de préciser de nombreux cas de figure courants, qu'il faut alors décrire dans le corps du texte. On ne peut pas formuler certaines règles simples, par exemple indiquer qu'un

élément A peut avoir au maximum 5 éléments fils B, sans même parler d'indiquer une fourchette 2-5.

On ne peut pas non plus exprimer des exclusions entre les attributs, ou indiquer que si tel attribut est présent, tel autre doit l'être aussi. On ne peut également pas indiquer que les valeurs de tel attribut, toutes balises comprises, doivent être uniques dans le document (cas flagrant, par exemple en HTML : l'attribut `id`). Bref, les DTD sont limitées.

C'est pourquoi on est passé à une autre syntaxe. Hélas, on a alors poussé jusqu'à l'extrême inverse : pour pouvoir tout décrire, on a créé une syntaxe si verbeuse, que la plupart des cas simples prennent de nombreuses lignes à représenter. Ce sont les schémas XML.

Naviguer dans un schéma XML

Les schémas XML ont pris la relève des DTD pour décrire formellement les possibilités de combinaison et d'inclusion d'éléments et d'attributs dans les langages à balises. Et quand on sait combien le W3C s'est épris des technologies XML (plus de vingt à ce jour, de XML lui-même à XSLT, de XPath à XML Query, de SVG à MathML...), on imagine facilement quel usage intensif est aujourd'hui fait des schémas XML.

Un schéma XML est lui-même un document XML. La syntaxe a été conçue pour couvrir tous les besoins imaginables. Et comme souvent dans de tels cas, elle aboutit à quelque chose de très épais pour les cas simples...

Prenons par exemple un fragment du schéma pour XHTML 1.1 Strict.

Listing C-5 Le schéma XML de l'élément form en XHTML 1.1

```
<xs:attributeGroup name="xhtml.form.attlist">
    <xs:attributeGroup ref="xhtml.Common.attrib"/>
    <xs:attribute name="action" type="xh11d:URI" use="required"/>
    <xs:attribute name="method" default="get">
        <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="get"/>
            <xs:enumeration value="post"/>
        </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="enctype" type="xh11d:ContentType"
                  ➥ default="application/x-www-form-urlencoded"/>
    <xs:attribute name="accept-charset" type="xh11d:Charsets"/>
    <xs:attribute name="accept" type="xh11d:ContentTypes"/>
```

```
</xs:attributeGroup>
<xs:group name="xhtml.form.content">
    <xs:sequence>
        <xs:choice maxOccurs="unbounded">
        <xs:group ref="xhtml.BlkNoForm.mix"/>
        <xs:element name="fieldset" type="xhtml.fieldset.type"/>
        </xs:choice>
    </xs:sequence>
</xs:group>
<xs:complexType name="xhtml.form.type">
    <xs:group ref="xhtml.form.content"/>
    <xs:attributeGroup ref="xhtml.form.attlist"/>
</xs:complexType>
```

À qui aurait le front de s'insurger que « c'est illisible ! », « c'est n'importe quoi ! », ou encore « beaucoup de bruit pour rien ! », je répondrais d'un air affable « vous avez bien raison ».

Seulement voilà, le W3C s'est amouraché des schémas XML et il n'est pas le seul, loin de là ! L'univers J2EE par exemple a basculé des DTD vers les schémas autour de l'apparition des Servlets 2.4 et de JSP 2.0, il y a déjà longtemps.

Parmi les concepts clés d'un schéma XML, il faut retenir qu'on définit le plus souvent séparément un type (une grappe d'éléments et de balises) et les noms des éléments qui reposent sur ces types (ce qui permet, il est vrai, une certaine factorisation, fort agréable). Les structures simples sont déclarées à l'aide de balises `<xs:simpleType>`, et les types complexes (en réalité, presque tous) au moyen de `<xs:complexType>`.

La notion de groupes, qui permet d'exprimer une exclusivité d'éléments ou d'attributs, ou de décrire des dépendances, est représentée par `<xs:group>` pour les éléments et `<xs:attributeGroup>` pour les attributs. Les éléments et attributs sont nommés respectivement avec `<xs:element>` et `<xs:attribute>`, les contraintes exprimées soit avec des attributs d'occurrence (`minOccurs` et `maxOccurs`), soit avec l'attribut `use`, ou encore avec des descriptions complexes `<xs:restriction>`. Le fragment présenté en listing C-5 est sympathique, parce qu'il illustre au moins un cas possible pour presque toutes ces syntaxes.

Il me faudrait un ouvrage entier pour vous présenter toutes les possibilités, mais si le sujet vous intéresse, vous trouverez une présentation assez détaillée et bien faite, pas à pas, sur http://www.w3schools.com/schema/schema_intro.asp.

Pour terminer, sachez qu'une syntaxe alternative fait de plus en plus d'aficionados, car tout en étant basée elle aussi sur XML, et bien que permettant d'exprimer tout ce qui est exprimable en schémas XML, elle est incomparablement plus simple et lisible. Il s'agit de Relax NG (nouvelle génération).

Relax NG est suffisamment populaire pour que la plupart des bibliothèques de traitement de grammaires XML la prennent aujourd'hui en charge (qu'on travaille en Java, C#, Delphi ou Ruby...). Toutefois, elle n'a pas encore su gagner les cœurs des grands organismes de standardisation. Pour vous faire une idée, consacrez donc quelques minutes à ce didacticiel sur le site officiel, qui permet de bien cerner la question : http://relaxng.org/tutorial-20011203.html. À titre de *teaser*, voici l'équivalent Relax NG du listing C-5.

Listing C-6 La spécification Relax NG de l'élément form en XHTML 1.1

```
<define name="form">
  <element name="form">
    <ref name="form.attlist"/>
    <oneOrMore>
      <ref name="Block.class"/>
    </oneOrMore>
  </element>
</define>

<define name="form.attlist">
  <ref name="Common.attrib"/>
  <attribute name="action">
    <ref name="URI.datatype"/>
  </attribute>
  <optional>
    <attribute name="method">
      <choice>
        <value>get</value>
        <value>post</value>
      </choice>
    </attribute>
  </optional>
  <optional>
    <attribute name="enctype">
      <ref name="ContentType.datatype"/>
    </attribute>
  </optional>
</define
```

Ayant ciblé l'exemple, on ne voit pas une immense différence de taille (28 lignes dans les deux cas), mais sentez-vous combien la seconde version est plus lisible que la première ?

Parcourir une RFC

Une RFC (*Request For Comments*) est un standard Internet élaboré par l'IETF (*Internet Engineering Task Force*). Je dis bien « Internet » et non « Web », car alors que le Web a vu le jour en 1992 avec la première page HTML, sous la houlette de Tim Berners-Lee, Internet existe lui depuis 1969 (même s'il s'appelait alors ARPANET).

Ce sont aujourd'hui des centaines de protocoles et de formats qui font fonctionner ce réseau mondial. Le grand public connaît les plus visibles : HTTP, POP, IMAP, SMTP, SSL, TLS, FTP... Ceux qui prêtent attention aux détails de leurs clients de messagerie connaissent peut-être aussi MIME. Il faut sans doute être dans l'informatique pour en connaître d'autres, comme SNMP, NTP, SSH, Telnet, RSTP et bien d'autres, sans parler des formats, de CSV à Atom.

Au total, ce sont plus de 4 600 documents standardisés qui ont déjà été émis, et avec environ 500 documents par an ces derniers temps, le phénomène ne fait qu'accélérer.

Format général d'une RFC

Le format d'une RFC a toutefois perduré à travers les âges et trahit aujourd'hui ses origines très modestes, à une époque où les imprimantes matricielles 8 points étaient le dernier cri, et où la notion même de réseaux d'ordinateurs n'était encore qu'une vision pleine d'espoir.

Le format principal d'une RFC est un fichier texte utilisant le jeu de caractères US-ASCII sur 7 bits (pas d'accents, pas de signes diacritiques, etc.). Le texte est formaté à 72 caractères de large. Certains termes ont un sens particulier, par exemple *must*, *should*, *can*, *must not*. Leur usage est défini par la RFC 2219. Une RFC est obligatoirement en anglais et le fichier porte l'extension `.txt`. On a encore bien des règles.

La RFC 2223 donne des détails sur le format ; une ébauche en plan depuis 2004 après 8 révisions, la 2223bis, met à jour certains points. On trouve même une RFC décrivant comment configurer MS Word pour produire un fichier RFC valide : la RFC 3285 !

Sachez par ailleurs qu'il existe trois sous-catégories de RFC : les standards de premier plan (STD), les pratiques recommandées (BCP, *Best Current Practice*) et les notes informatives (FYI, *For Your Information*). Dernier point : les RFC publiées le 1er avril sont systématiquement des canulars. Faites donc une recherche, ça vaut le détour.

Enfin, le format texte d'une RFC rend difficile la navigation à l'intérieur du document, les références n'étant pas des hyperliens. Aussi, sachez qu'il existe un accès HTML aux documents, dont la navigation est par conséquent facilitée. Pour obtenir la version HTML d'une RFC, c'est facile : prenez l'URL officielle de la version principale, par exemple : http://www.ietf.org/rfc/rfc4287.txt.

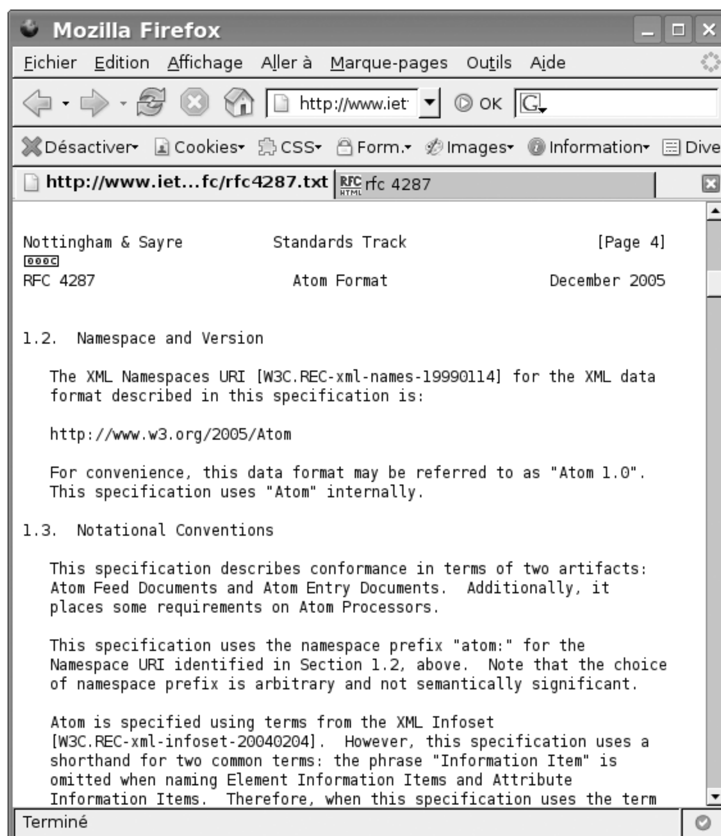


Figure C–6
L'aspect texte brut de la RFC du format Atom

Remplacez le « www. » par « `tools.` », le chemin `/rfc/` par `/html/` et retirez l'extension. Vous obtenez ceci : http://tools.ietf.org/html/rfc4287.

Et voilà une version HTML de votre RFC, avec des liens automatiques pour les numéros de pages, les URL, les références et les renvois entre sections. Par ailleurs, certaines portions sont mises en gras (titres) ou en italique, et les en-têtes et pieds de page sont grisés. Jugez plutôt (figure C–7).

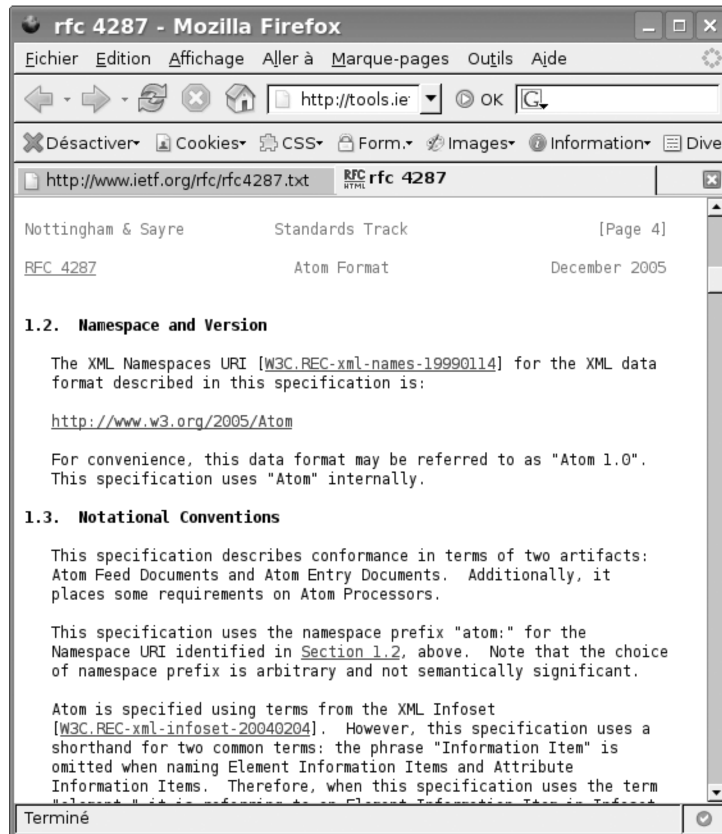


Figure C–7
La RFC passée à la moulinette HTML

Structure générale d'une RFC

Une RFC commence toujours par son en-tête déclaratif, repris en en-tête de chaque page. Voici un exemple, raccourci en largeur pour tenir sur cette page :

```
Network Working Group                          M. Nottingham, Ed.
Request for Comments: 4287                          R. Sayre, Ed.
Category: Standards Track                           December 2005
```

On a sur la gauche le nom du groupe de travail (l'IETF en compte un certain nombre), le statut (RFC) et le numéro du document. Les RFC sont en effet le plus souvent référencées par leur numéro. La catégorie indique plus précisément le statut. Ici, le terme *Standards Track* confirme que le document a valeur de standard.

Sur la droite, on trouve la liste des auteurs principaux (le « `Ed.` » signifie *Editor*), et la date de publication du document, avec le mois et l'année.

L'en-tête des pages suivantes reprendra toutes ces informations ainsi que le numéro de page et le titre du document :

```
Nottingham & Sayre        Standards Track              [Page 1]

RFC 4287                    Atom Format            December 2005
```

Une RFC comporte obligatoirement une introduction, dont les premiers paragraphes doivent décrire avec concision le contexte et l'objectif du standard. L'IETF est ici généralement plus efficace que le W3C dans ses *abstracts*...

L'introduction comprend obligatoirement, généralement sur la fin, une section du type *Notational Conventions*, qui détaille les notations spécifiques au document, et précise systématiquement le sens formel de termes comme *must*, *should*, *cannot*, etc. en faisant une référence à leur définition exacte dans la RFC 2119.

Une RFC se clôt généralement par une liste de références normatives (autres standards) ou informatives, une liste des contributeurs au standard (auteurs n'ayant pas un statut « principal »), et souvent des versions consolidées de grammaires formelles ou autres informations techniques fragmentées dans le corps du document.

Il n'est pas rare de voir, dans les derniers chapitres d'une RFC, un *IANA Considerations*, si le standard entraîne le dépôt de nouveaux types MIME ou réquisitionne certains numéros de ports réseau, ainsi qu'un *Security Considerations*, qui détaille toute faille de sécurité potentielle envisagée par les auteurs.

Enfin, les RFC font une utilisation intensive de la syntaxe EBNF, évoquée plus haut. Prenez par exemple la RFC 2616 (http://tools.ietf.org/html/rfc2616), celle qui décrit HTTP/1.1. Les syntaxes des en-têtes de requête et de réponse sont décrites en EBNF. Voici un extrait dans le listing suivant.

Listing C-7 Syntaxe EBNF de l'en-tête de requête Accept en HTTP/1.1

```
Accept         = "Accept" ":"
                 #( media-range [ accept-params ] )

media-range    = ( "*/*"
                 | ( type "/" "*" )
                 | ( type "/" subtype )
                 ) *( ";&quot; parameter )
accept-params = ";&quot; "q" "=" qvalue *( accept-extension )
accept-extension = ";&quot; token [ "=" ( token | quoted-string ) ]
```

Comme souvent, l'ensemble de la syntaxe est expliquée clairement en début de RFC, dans la section *Notational Conventions*, pour ne pas perdre les lecteurs.

Notez que la RFC citée en exemple plus haut, celle du format de flux Atom, innove en utilisant des schémas Relax NG pour décrire son balisage.

Vos spécifications phares

En tant que développeur web, voici les principales spécifications qui devraient vous intéresser :

HTML 4.01 (éléments et attributs autorisés)

- Référence : http://www.w3.org/TR/html4/
- Version française : http://www.la-grange.net/w3c/html4.01/cover.html

XHTML 1.0 (repose sur HTML 4.01)

- Référence : http://www.w3.org/TR/xhtml1/
- Version française : http://www.la-grange.net/w3c/xhtml1/

CSS 2.1

- Référence : http://www.w3.org/TR/CSS21/
- Version française (2.0 !) : http://www.yoyodesign.org/doc/w3c/css2/cover.html

Attention, la version 2.1 a beaucoup modifié la version 2.0...

JavaScript 1.5

- DevMo : http://developer.mozilla.org/fr/docs/JavaScript
- ECMA : http://www.ecma-international.org/publications/standards/Ecma-262.htm

DOM niveau 2

- Noyau : http://www.w3.org/TR/DOM-Level-2-Core/
- HTML : http://www.w3.org/TR/DOM-Level-2-HTML/
- Événements : http://www.w3.org/TR/DOM-Level-2-Events/
- Style : http://www.w3.org/TR/DOM-Level-2-Style/

D

Développer avec son navigateur web

C'est à ses outils qu'on reconnaît le bon artisan... Il est effarant de constater combien, encore aujourd'hui, les développeurs web persistent à sous-utiliser leurs navigateurs dans leurs développements. Le navigateur est relégué au rang de visualiseur, ce qui est insensé !

Il ne s'agit pas seulement de savoir configurer le cache pour être certain de toujours utiliser la dernière version d'une ressource qui change fréquemment. Les principaux navigateurs savent explorer les méandres internes de la page, son DOM, ses CSS, son accessibilité... Ils fournissent des pelletées d'outils informatifs, d'analyseurs, et même des débogueurs JavaScript !

Il ne tient qu'à vous d'ajouter une bonne dose de confort et d'efficacité, en somme, de productivité, à votre méthodologie de développement. Si j'étais vous, je lirais attentivement cette annexe avant tout le reste.

Le cache peut être votre pire ennemi

Le cache peut être votre pire ennemi, surtout alors que vous suivez les exemples de ce livre. Le cache, c'est très pratique : il nous évite de passer notre vie à recharger tout le temps des images, feuilles de styles, etc. Mais en développement, cela peut poser problème. On change fréquemment la CSS, le JavaScript, les feuilles XSLT...

Chaque navigateur a sa façon bien à lui de gérer ce cache, qu'il s'agisse du disque ou de la mémoire. Certains seront plus réactifs aux changements de ressources auxquelles on accède directement que pour celles obtenues par Ajax ; d'autres rechargeront plus volontiers du JavaScript que des CSS, etc.

Le rafraîchissement strict

La plupart des navigateurs fournissent un mécanisme clavier pour effectuer un rafraîchissement strict, c'est-à-dire un rechargement effectif de l'intégralité des ressources utilisées par la page :

- Mozilla, Camino, Firefox, MSIE : au lieu de presser simplement F5 ou de cliquer sur le bouton idoine, maintenez la touche Ctrl ou Cmd enfoncée pendant ce temps. Hors MSIE, vous pouvez aussi utiliser Maj avec le bouton de rechargement ou F5.
- Safari : utilisez Cmd+Maj+R au lieu de Cmd+R, ou maintenez Cmd enfoncée en pressant le bouton de rafraîchissement.
- Opera : il n'y a pas de mécanisme de rafraîchissement strict ! C'est ahurissant, mais c'est comme ça. En théorie, Opera ignore le cache à chaque rafraîchissement explicite avec F5 ou son bouton de rechargement. Dans la pratique, sur les exemples de ce livre, j'ai dû configurer le cache. On peut donc décider de faire de même, pour qu'il vérifie soigneusement les nouvelles versions des ressources (voir un peu plus bas) ou vider le cache complètement (voir ci-dessous).
- Konqueror : ignore le cache à chaque rechargement explicite avec F5 ou son bouton de rechargement.

Vider le cache

Il est parfois nécessaire de vider complètement le cache, ou en tout cas les parties du cache associées à la page en cours. Voici comment faire:

- Firefox : allez dans les options (Outils>Options sous Windows, Édition> Préférences sous Linux, Firefox>Préférences sous Mac OS X) et choisissez la catégorie Vie privée puis l'onglet Cache. Cliquez sur le bouton Vider le cache. Depuis la 1.5, Firefox permet également de supprimer instantanément tout ou partie de votre état « privé » en enfonçant Ctrl+Maj+Suppr ou en allant dans Outils>Effacer mes traces.

Attention, par défaut cela sélectionne bien plus que le cache ! Vous pouvez le configurer depuis la catégorie Vie privée des options, avec le bouton Paramètres.

- Mozilla : allez dans Édition>Préférences>Cache et choisissez Vider le cache.
- MSIE : allez dans Outils>Options Internet>Général>Fichiers Internet temporaires. Le bouton Supprimer les fichiers... permet de vider le cache (pensez à cocher la case Supprimer tout le contenu hors-ligne). Validez.
- Safari : dans le menu Safari, choisissez Vider le cache..., ou pressez Cmd+Option+E.
- Opera : allez dans Outils>Préférences>Avancé>Historique et choisissez le bouton Vider maintenant.
- Konqueror : allez dans Configuration>Configurer Konqueror. Dans la liste à gauche, faites défiler pour choisir la catégorie Cache. Cliquez sur le bouton Vider le cache.

Configurer le cache

Pour éviter les difficultés avec le cache, il est important de le configurer correctement. En développement, le mieux est de lui demander de vérifier à chaque requête (à l'aide d'une requête HTTP HEAD) si chaque ressource a changé depuis sa dernière version. On peut parfois le désactiver complètement, mais c'est moins utile.

Voici les modes opératoires :

- Mozilla, Camino, Firefox : c'est une mauvaise idée, mais vous pouvez demander au navigateur d'allouer 0 Ko au cache dans les options. Toutefois, il ne s'agit que du cache disque : un cache mémoire existe tout de même. Il est désactivable via les options « expert » de `about:config`. Là aussi, c'est une mauvaise idée...
- MSIE : allez dans Outils>Options Internet>Général>Fichiers Internet temporaires. Cliquez sur Paramètres... Choisissez l'option À chaque visite de la page. Ainsi, MSIE vérifiera à chaque fois et récupérera toute nouvelle version.
- Safari : à n'utiliser qu'à vos risques et périls... Fermez totalement Safari, ouvrez un terminal (outil Terminal dans Applications>Utilitaires), et tapez deux lignes : d'abord `rm -fr ~/Library/Caches/Safari`, puis `touch ~/Library/Caches/Safari`. Quittez le terminal.
- Opera : allez dans Outils>Préférences>Avancé>Historique et configurez les listes déroulantes Vérifier les documents et Vérifier les images, en les définissant à Toujours. Vous pouvez aussi choisir de désactiver le cache en définissant Cache mémoire et Cache disque à Off, mais ce n'est pas une très bonne idée...
- Konqueror : allez dans Configuration>Configurer Konqueror. Dans la liste à gauche, faites défiler pour choisir la catégorie Cache. Si la case Utiliser le cache est cochée, assurez-vous que l'option Assurer la synchronisation du cache est sélectionnée. Pour désactiver complètement le cache, décochez simplement la case. Mais sur Konqueror, c'est vraiment une mauvaise idée : sa gestion du cache est bien adaptée au développement.

Firefox, favori du développeur grâce aux extensions

De base, Firefox fournit une console JavaScript plutôt correcte (Outils>Console JavaScript), mais pas exceptionnelle. Nous l'avons vue au chapitre 2.

Firefox est néanmoins sans doute le meilleur navigateur pour développer des applications web, tant l'univers de ses extensions est riche d'outils fabuleux ! Je n'en cite que deux qui me semblent incontournables : la Web Developer Toolbar de Chris Pederick et Firebug de Joe Hewitt.

Nous avons déjà vu Firebug en détail aux chapitres 2, 3 et 4. Petite extension légère, elle fournit néanmoins une console JavaScript efficace, un objet `console` utilisable dans vos scripts, un pisteur de requêtes Ajax, un débogueur JavaScript largement suffisant, et un inspecteur multivue (code source, très utile ; DOM, plus classique ; Layout, parfois irremplaçable...). C'est un bijou.

La Web Developer Toolbar de Chris Pederick est devenue une véritable légende. Au point que la très officielle Internet Explorer Developer Toolbar est une copie évidente. Il me faudrait un chapitre entier pour la décrire, je vous conseille plutôt de consulter son site, de la télécharger, à moins que ce ne soit déjà fait, et de vous laisser convaincre par vous-même !

Ceci dit, vous trouverez énormément d'extensions et d'outils complémentaires, qui correspondent peut-être davantage à vos besoins. Je cite rapidement le débogueur JavaScript officiel : Venkman ; l'inspecteur DOM à activer à l'installation de Firefox ; mais aussi la XML Developer Toolbar, GreaseMonkey, Platypus, Aardvark...

Les URL correspondantes :

- Web Developer Toolbar : http://chrispederick.com/work/webdeveloper/
- Firebug : http://www.joehewitt.com/software/firebug/
- XML Developer Toolbar : https://addons.mozilla.org/firefox/2897/
- GreaseMonkey : http://greasemonkey.mozdev.org/
- Platypus : http://platypus.mozdev.org/
- Aardvark : http://karmatics.com/aardvark/
- Venkman : http://www.mozilla.org/projects/venkman/

Les trésors du menu Debug caché dans Safari

Safari dispose d'un menu `Debug` caché (par défaut), qui regorge d'options intéressantes pour le développeur web. Pour l'activer, il suffit de réaliser les manipulations suivantes :

1. Fermez totalement Safari.
2. Ouvrez un terminal (Applications>Utilitaires>Terminal).
3. Tapez `defaults write com.apple.Safari IncludeDebugMenu 1`.
4. Vous pouvez relancer Safari.

Bon, il n'est pas localisé en français. Mais regardez donc :



Figure D–1
Le menu Debug de Safari

Il donne envie, non ?

MSIE et la Internet Explorer Developer Toolbar

MSIE ne brille guère par ses possibilités natives. Pas de console JavaScript, une gestion des plus pénibles pour les erreurs et avertissements JavaScript... Toutefois, on peut améliorer la situation avec deux outils : le **débogueur de script** et la **Internet Explorer Developer Toolbar**.

Le débogueur de script est téléchargeable ici :

http://www.microsoft.com/downloads/details.aspx?familyid=2f465be0-94fd-4569-b3c4-dffdf19ccd99 (oui, c'est une URL très longue et impossible à taper facilement...)

On trouve des instructions d'utilisation sur :

http://msdn.microsoft.com/library/en-us/sdbug/Html/sdbug_1.asp

La Internet Explorer Developer Toolbar est très fortement inspirée de la Web Developer Toolbar de Chris Pederick (voir pour s'en convaincre http://www.thinklemon.com/weblog/stuff/WebDevelopervsDeveloperToolbar.jpg). On la télécharge, comme d'habitude, via une URL ahurissante :

http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038

C'est assez complet et ça ajoute des mini-explorateurs DOM, CSS, etc. C'est mieux que rien.

Et Opera, qu'a-t-il pour nous ?

Opera est extrêmement riche en fonctionnalités, notamment en ce qui concerne l'accessibilité. Mais côté développement, il n'offre pas grand-chose.

Il faut reconnaître que sa fenêtre de messages est excellente : elle concentre les messages simples, d'avertissement et d'erreur pour un grand nombre de domaines (CSS, XML, JavaScript, DOM, HTML, SVG, etc.), le tout filtrable facilement. Mais on manque de débogueur JavaScript, d'inspecteur DOM, etc.

Opera n'est pas vraiment le navigateur du développement. Mais il faut tout de même tester vos sites avec !

Index

D

E

R

S